Cache-Conscious Performance Optimization for Similarity Search

Maha Alabduljalil, Xun Tang, Tao Yang
Department of Computer Science
University of California, Santa Barbara
{maha,xtang,tyang}@cs.ucsb.edu

ABSTRACT

All-pairs similarity search can be implemented in two stages. The first stage is to partition the data and group potentially similar vectors. The second stage is to run a set of tasks where each task compares a partition of vectors with other candidate partitions. Because of data sparsity, accessing feature vectors in memory for runtime comparison in the second stage, incurs significant overhead due to the presence of memory hierarchy. This paper proposes a cache-conscious data layout and traversal optimization to reduce the execution time through size-controlled data splitting and vector coalescing. It also provides an analysis to guide the optimal choice for the parameter setting. Our evaluation with several application datasets verifies the performance gains obtained by the optimization and shows that the proposed scheme is upto 2.74x as fast as the cache-oblivious baseline.

Categories and Subject Descriptors

H.3.3 [**Information Search and Retrieval**]: Search Process, Clustering; H.3.4 [**Systems and Software**]: Performance evaluation

Keywords

Similarity search, data traversal, memory hierarchy

1. INTRODUCTION

All Pairs Similarity Search (APSS) [6], which identifies similar objects among a given dataset, has many important applications. For example, collaborative filtering provides recommendations by determining which users have similar tastes [29, 7], search query suggestions identifies queries with similar search results [22], web mirrors and plagiarism recognition [25], coalition detection for advertisement frauds [20], query suggestions [22], spam detection [8, 16, 14], clustering [5], and finally near duplicate detection [12, 30].

The complexity of a naïve APSS can be quadratic to the dataset size. Previous research on expediting similarity computing, developed filtering methods to eliminate unnecessary computations [6, 28, 2], applied inverted indexing to compare vectors only when they share features [18, 21], and used partitioning and parallelization techniques [1]. The LSH based mapping can approximately map vectors to the same bucket when they are potentially similar. However, none of the previously developed methods have considered the impact of memory hierarchy on execution time. The main memory access latency can be 10 to 100 times slower than the L1 cache latency. Thus, the unorchestrated slow memory access can significantly impact performance.

In this paper, we exploit memory hierarchy and develop orthogonal techniques to improve the efficiency of APSS by optimizing data layout and traversal methods. Specifically, we investigate how data traversal affects the use of memory layers. This method is also motivated by the work in query processing [23] and sparse matrix computation that considers cache optimization in computation arrangement.

Similarity comparisons can be performed through a number of tasks where each of them compares a partition of vectors with other candidate vectors [1].

We propose two algorithms PSS1 and PSS2 to exploit the memory hierarchy explicitly. PSS1 splits the data hosted in the memory of each task to fit into the processor's cache and PSS2 coalesces data traversal based on a length-restricted inverted index. We provide an analytic cost model and identify the parameter values that optimize the performance. Hence, the contribution of this paper is a memory-hierarchy aware framework for fast similarity comparison with optimized data layout and traversal.

The rest of this paper is organized as follows. Section 2 reviews background and related work. Section 3 discusses the design framework and PSS1 algorithm for cache-aware data splitting. Section 4 analyzes cost model of PSS1 and demonstrates the impact of the parameters on memory access performance. Section 5 presents the optimization using vector coalescing named PSS2. Section 6 is our experimental evaluation that assess PSS1 and PSS2. Section 7 concludes this paper.

2. BACKGROUND AND RELATED WORK

Following the definition in [6], the APSS problem is defined as follows. Given a set of vectors $d_i = \{w_{i,1}, w_{i,2}, \cdots, w_{i,m}\}$, where each vector contains at most m non-negative features and is normalized to a unit length, then the cosine-based similarity between two vectors is computed as:

$$Sim(d_i, d_j) = \sum_{t \in (d_i \cap d_j)} w_{\mathrm{i,t}} \times w_{\mathrm{j,t}}.$$

Two vectors x, y are considered similar if their similarity score exceeds a threshold τ, namely $Sim(x, y) \geq \tau$. The time complexity of APSS is high, especially for a big dataset. There are application-specific methods applied to reduce the complexity. For example, text mining removes stop-words or extremely high frequent features [18]. We use such preprocessing throughout the experiments in Section 6. Generally, there are two groups of optimization techniques developed in the previous work to accelerate APSS.

- **Dynamic computation filtering**. Partially accumulated similarity scores can be monitored at runtime and dissimilar pair of documents can be detected dynamically based on the given similarity threshold without the complete derivation of the total similarity value [6, 28, 21] .
- **Similarity-based grouping in data preprocessing**. The search scope for similarity can be reduced when potentially similar vectors are placed in one group. One approach is to use inverted indexing [28, 18, 21] developed for information retrieval [5]. This approach identifies vectors that share at least one feature, as potentially similar. Hence, it skips unnecessary data traversal while conducting APSS. Another approach is LSH that can approximately map similar vectors into one group [11, 26]. This approach has a tradeoff between precision, recall ratio, and redundant computations when using multiple hash functions. The study in [2] shows that exact comparison algorithms can deliver performance competitive to LSH when computation filtering are used.

  Another approach is partition-based search [1] which statically identifies dissimilar vectors to guide data grouping. Runtime computation avoids the comparison among dissimilar vectors.

Cache optimization for computationally intensive applications is studied in the context of general database query processing [23, 19] and matrix-based scientific computing [10, 9, 27, 4]. Motivated by these studies, we investigate the opportunities of cache-conscious optimization targeting APSS.

3. FRAMEWORK AND CACHE-AWARE DATA SPLITTING

In this section, we give an overview of the partition-based comparison framework [1] and then present the caching optimization strategies as our contributions.

3.1 Framework and basic algorithm

The framework for partition-based similarity search (PSS) consists of two steps. The first step is to divide a dataset into a set of partitions. During this process, the dissimilarity relationship among partitions is identified so that unnecessary comparisons among them are avoided. The second step is to assign a partition to a task and each task compares this partition with other potentially similar partitions.

Dissimilarity-based partitioning identifies dissimilar vectors as much as possible without explicitly computing the product of their features. One approach is to use the following inequality that uses the 1-norm and ∞-norm of each vector.

$$Sim(d_i, d_j) \leq min(||d_i||_\infty ||d_j||_1, ||d_j||_\infty ||d_i||_1) < \tau.$$

The partitioning algorithm sorts the vectors based on their 1-norm values and uses the sorted list to identify dissimilar pairs $(d_i,\ d_j)$ in complexity $O(n \log n)$ satisfying the inequality $\|d_i\|_1 < \frac{\tau}{||d_j||_\infty}$. A different τ value would affect the outcome of the dissimilarity detection in the above partitioning. The details for the above static partitioning is discussed in [1]. This paper focuses on optimizing the task execution after the static partitioning is applied. Note that this scheme can also be applied with LSH when approximations are allowed. When vectors are mapped into a set of dissimilar buckets using LSH, some buckets can still be big and the static partitioning can be further applied to divide such buckets.

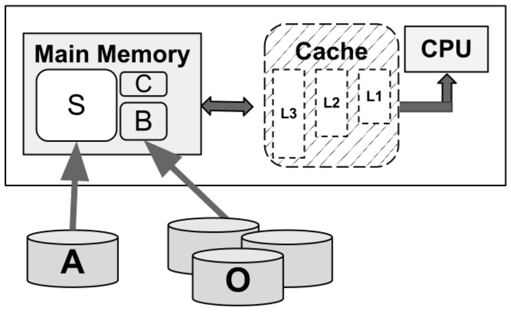


Figure 1: A PSS task compares the assigned partition A with other partitions O.

Figure 1 depicts a task for partition-based similarity search interacting with a CPU core with multiple levels of cache. Two or three cache levels are typical in today's Intel or AMD architecture [17, 15]. We assume that the assigned partition A fits the memory of one machine as the data partitioning can be adjusted to satisfy such an assumption. But vectors of O can exceed memory and need to be fetched gradually from a local or remote storage. In a computer cluster with the distributed file system such as Hadoop, a task can seamlessly fetch data from the file system without worrying about the machine location of data.

The memory used by each task has three areas, as illustrated in Figure 1. 1) Area S: hosts the assigned partition A. 2) Area B: stores a block of vectors fetched from other candidate partitions O at each comparison step. 3) Area C: stores intermediate results temporarily.

Figure 2 describes the function of a PSS task. Each task loads the assigned vectors, whose data structure is in forward index format, into area S. Namely, each vector consists of an ID along with a list of feature IDs and their corresponding weights, stored in a compact manner. After loading the assigned vectors, the task inverts them locally within area S. It then fetches a number of vectors from O, in forward index format, and place them into area B.

Let d_j be the vector fetched from O to be processed (Line 5). For each feature t in d_j, PSS uses the inverted index in area S to find the localized t's posting (Line 10). Then weights of vector d_i from t's posting and d_j contribute a partial score towards the final similarity score between d_j and d_i. After all the features of d_j are processed, the similarity scores between d_j and the vectors in S are validated (Line 17) and only those that exceed the threshold are written to disk. In COMPARE(S, d_j), the dissimilarity of vector d_i in S with d_j can be marked (Line 14) by using a negative value

```
TASK (A, O)
1: Read all vectors from assigned partition $A$ into $S$
2: Build inverted index of these vectors and store in $S$
3: repeat
4:   Fetch a set of vectors from $O$ into $B$
5:   for $d_j \in B$ do
6:     COMPARE ($S$, $d_j$)
7: until all vectors in $O$ are fetched

COMPARE (S, $d_j$)
8: Initialize array score of size |S| with zeros
9: $r_j \leftarrow ||d_j||_1$
10: for $t \in d_j$ And $Posting(t) \in$ S do
11:   for $d_i \in Posting(t)$ and $d_i$ is a candidate do
12:     $score[i]{=}score[i]{+}w_{i,t}{\times}w_{j,t}$
13:     if ($score[i]{+}maxw[d_i]{\times}r_j{<}\tau$) then
14:       Mark $d_i$ as non-candidate
15:   $r_j = r_j - w_{j,t}$
16: for $i = 1$ to |S| do
17:   if $score[i]{>}\tau$ then
18:     Write $(d_i, d_j, score[i])$
```

Figure 2: PSS task.

for *score*[*i*]. Array *maxw*[ ] contains the ∞-norm value of vector d_i.

3.2 Cache-conscious data splitting

When dealing with a large dataset, the number of vectors in each partition is high. Having a large number of vectors increase the benefits of using inverted indexing as shown in Figure 2. But it has a problem that the accessed areas S or C may not fit in the fast cache. In that case, temporal locality is not exploited, meaning the second access of the same element during any computation will be a cache miss. As we show in the next section, this leads to frequent slow memory access and a significant increase in execution time. Since fast access of each area S, B or C is equally important in the core computation (Lines 12 and 13), one idea is to let area C fit in L1 cache by explicitly dividing vectors of the assigned partition in S into a set of splits and have the task focus on one split at a time.

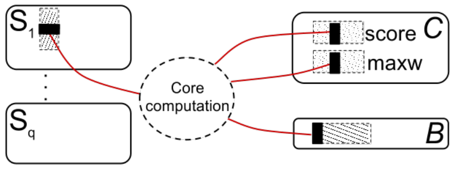


Figure 3: A partition in area S is further divided into multiple splits for each PSS1 task. Four data items are involved in the core computation. The striped area indicates cache coverage.

Figure 3 illustrates this cache-conscious data splitting idea. The corresponding algorithm called PSS1 is shown in Figure 4. First, it divides the hosted vectors in S into q splits. Each split S_i is of size s. PSS1 then executes q comparison sub-tasks. Each sub-task compares vectors from S_i with a vector b_j in B. The access in area C is localized such that array *score*[ ] and *maxw*[ ] can fully fit in L1 cache. This improves temporal locality of data elements for area C and

```
TASK (A, O)
1: Read and divide $A$ into $q$ splits
2: Build an inverted index for each split $S_i$ and store in $S$
3: repeat
4:   Fetch a set of vectors from $O$ into $B$
5:   for $d_j \in B$ do
6:     for $S_i \in S$ do
7:       COMPARE ($S_i$, $d_j$)
8: until all vectors in $O$ are fetched
```

Figure 4: PSS1 task.

reduces the access time by an order of magnitude. The core computation speeds up as a result.

The data splitting also introduces potential benefits from exploiting the multi-core CPU architecture via threads. Every time a data block from O is fetched into B, there can be multiple threads running in parallel to execute COMPARE(S_i, d_j) where d_j is a vector in B.

The question is, how to determine the s value of each split so that the caches are best utilized? This is discussed next.

4. COST ANALYSIS AND CACHE PERFORMANCE OF PSS1

We model the total execution time of each PSS1 task and analyze how memory hierarchy affects the running time. This analysis facilitates the identification of optimized parameter setting. Table 1 describes the parameters used in our analysis. They represent the characteristics of the given dataset, algorithm variables, and the system setting.

Dataset	
$w_{d,t}$	Weight of feature t in vector d
τ	Similarity threshold
k	Average number of non-zero features in d
Algorithm	
S, B, C	Memory usage for each task
n	Number of vectors to compare per task ($\|O\|$)
s	Avg. number of vectors for each split in S
b	Number of vectors fetched and stored in B
S_i	A split in area S divided by PSS1
q	Number of splits in S
h	Cost for t-posting lookup in table
δ_{total}	Cost of accessing the hierarchical memory
$m_j(X)$	Miss ratio in level j cache for area X
$D_j(X)$	Number of misses in level j cache for area X
D_j	Total number of access misses in level j cache
p_s	Average posting length in the inverted index of each split S_i
p_b	Average posting length in the inverted index of b vectors in B
Infrastructure	
l	Cache line size
e_s, e_b, e_c	Element size in S, B, C respectively
f_s, f_b, f_c	Effective prefetch factor for elements in S, B and C respectively
$\delta_1, \delta_2, \delta_3$	Latency when accessing L1, L2, and L3 cache
δ_{mem}	Latency when accessing main memory
ψ	Cost of addition and multiplication

Table 1: Modeling Symbols

4.1 Task execution time

The total execution time for each task contains two parts: I/O and computation. I/O cost occurs for loading the assigned vectors A, fetching other potentially similar vectors, and writing similarity pairs to disk storage. Notice that in fetching other vectors for comparison, the algorithm always fetches a block of vectors to amortize the startup cost of I/O. For the datasets we have used, read I/O takes about 2% of total cost while write I/O takes about 10-15%. Since I/O cost is the same for the baseline PSS and our proposed schemes, we do not model it in this paper.

For each split, the computation time contains a small overhead for the index inversion of its s vectors. Because the inverted index is built once and reused every time a partition is loaded, this part of computation becomes negligible and the comparison time with other vectors dominates. The core part (Lines 12, 13 in Figure 2) is computationally intensive. Following Table 1, h is the cost of looking up the posting of a feature appeared in a vector in B. Symbol p_s is the average length of postings visited in S_i (only when a common feature exists), so it estimates the number of iterations for Line 10. Furthermore, there are 4 memory accesses in Line 12 and 13, regarding data items $score[i]$, $w_{i,t}$, $w_{j,t}$, and $maxw[d_i]$. Other items, such as r_j, and τ, are constants within this loop and can be pre-loaded into registers. There are 2 pairs of multiplication and addition involved (one in Line 12 and one in Line 13) bringing in a cost of 2ψ. For simplicity of the formula, we model the worst case where none of the computations are dynamically filtered.

For a large dataset, the cost of self-comparison within the same partition for each task is negligible compared to the cost of comparisons with other vectors in O. The execution time of PSS1 task (Figure 4) can be approximately modeled as follows.

$$\text{Time} = q\left[nk(\overbrace{h}^{lookup} + \overbrace{p_s \times 2\psi}^{multiply+add}) + \overbrace{\delta_{total}}^{traverse\ S,B,C} \right]. \quad (1)$$

As s increases, q decreases and the cost of inverted index lookup may be amortized. In the core computation, p_s increases as s increases. More importantly, the running time can be dominated by δ_{total} which is the data access cost due to cache or memory latency. The data access cost is affected by s because of the presence of memory hierarchy. We investigate how to determine the optimal s value to minimize the overall cost in the following subsection.

4.2 Memory and Cache Access of PSS1

Here, we estimate the cost of accessing data in S, B, and C. Define D_0 as the total number of data accesses in performing COMPARE(S_i, d_j) in Figure 4. Define D_j as the total number of data access misses in cache level j. δ_i is the access time at cache level i. δ_{mem} is the memory access time.

$$\begin{aligned}\delta_{total} = &(D_0 - D_1)\delta_1 + (D_1 - D_2)\delta_2 + (D_2 - D3)\delta_3 \\ &+ D_3\delta_{mem}.\end{aligned} \quad (2)$$

To conduct the computation in Lines 12 and 13 of Figure 2, the program needs to access weights of S_i, B, $score[\,]$ and $maxw[\,]$ in C. We model these accesses separately then add them together as follows:

$$D_0 = D_0(S_i) + D_0(B) + D_0(C) = \overbrace{nkp_s}^{S_i} + \overbrace{nk}^{B} + \overbrace{2nkp_s}^{C}. \quad (3)$$

Case		S_i	C	Description
(1)	m_1	$\frac{e_s}{f_s l}$	0	C fits in L1; S_i does not fit L1, but fits in L2.
	m_2	0	0	
	m_3	0	0	
(2)	m_1	$\frac{e_s}{f_s l}$	$\frac{e_c}{f_c l}$	S_i and C do not fit in L1, but fit in L2.
	m_2	0	0	
	m_3	0	0	
(3)	m_1	$\frac{e_s}{f_s l}$	$\frac{e_c}{f_c l}$	C does not fit in L1, but fits in L2; S_i does not fit in L2 but fits in L3.
	m_2	1	0	
	m_3	0	0	
(4)	m_1	$\frac{e_s}{f_s l}$	$\frac{e_c}{f_c l}$	S_i and C do not fit in L2, but fit in L3.
	m_2	1	1	
	m_3	0	0	
(5)	m_1	$\frac{e_s}{f_s l}$	$\frac{e_c}{f_c l}$	C does not fit in L2 but fits in L3; S_i does not fit in L3.
	m_2	1	1	
	m_3	1	0	
(6)	m_1	$\frac{e_s}{f_s l}$	$\frac{e_c}{f_c l}$	S_i and C do not fit in L3.
	m_2	1	1	
	m_3	1	1	

Table 2: Cases of cache miss ratios for split S_i and area C in PSS1 at different cache levels. Column 3 is the cache miss ratio $m_j(S_i)$ for accessing data in S_i. Column 4 is the cache miss ratio $m_j(\mathcal{C})$ for accessing data in $\mathcal{C}$. Column 5 describes the condition of each case.

Define $D_j(X)$ as the total number of data accesses missed in cache level j for accessing area X. $m_j(X)$ is the cache miss ratio to access data for area X in cache level j.

$$\begin{aligned}D_j =&D_j(S_i) + D_j(B) + D_j(C) \\ =&D_{j-1}(S_i) * m_j(S_i) + D_{j-1}(B) * m_j(B) \\ &+ D_{j-1}(C) * m_j(C).\end{aligned} \quad (4)$$

Table 2 lists six cases of miss ratio values $m_j(S_i)$ and $m_j(C)$ at different cache levels j. The miss ratio for B is not listed and is considered close to 0 assuming it is small enough to fit in L1 cache after warm-up. That is true for our tested datasets. For a dataset with long vectors and B cannot fit in L1, there is a small overhead to fetch it partially from L2 to L1. Such overhead is negligible due to the relative small size of B, compared to S_i and C. We explain this table in more details from the following aspects.

- A cache miss triggers the loading of a cache line from next level. We assume the cost of a cold cache miss during initial cache warm-up is negligible and the cache replacement policy is LRU-based. Thus the cache miss ratio for consecutive access of a vector of elements is $\frac{1}{l/e}$ where l is the cache line size and e is the size of each element in bytes. We assume that cache lines are the same in all cache levels for simplicity, which matches the current Intel and AMD architecture.

- The computer system prefetches a few cache lines in advance, in anticipation of using consecutive memory regions [17, 15]. Let f_s be the effective prefetch factor for S_i, and e_s be the element size for S_i. The cache miss ratio for accessing S_i is adjusted as $\frac{e_s}{f_s l}$. Similarly, the cache miss ratio for accessing C is adjusted as $\frac{e_c}{f_c l}$

and the cache miss ratio for accessing B is adjusted as $\frac{e_b}{f_b l}$.

- In Case (1), s is small. C can fit in L1 cache. Thus after initial data loading, its corresponding cache miss ratios $m_1(C_1)$, $m_2(C_1)$, and $m_3(C_1)$ are close to 0. Then $m_1(S_i) = \frac{e_s}{f_s l}$, and $m_2(S_i)$ and $m_3(S_i)$ are approximately 0 since each split can fit in L2 (but not L1). In this case, s is too small, the benefit of using the inverted index does not outweigh the overhead of the inverted-index constructions and dynamic look-up.

- In Case (2), S_i and C can fit in L2 cache (but not L1). $m_1(S_i) = \frac{e_s}{f_s l}$, and $m_1(C) = \frac{e_c}{f_c l}$. $m_2(S_i)$ and $m_3(S_i)$ are approximately 0. Thus δ_{total} is:

$$\delta_{total} = \left[nkp_s(1 - \frac{e_s}{f_s l}) + nkp_s + 2nkp_s(1 - \frac{e_c}{f_c l}) \right] \delta_1 + \left[nkp_s \frac{e_s}{f_s l} + 2nkp_s \frac{e_c}{f_c l} \right] \delta_2. \quad (5)$$

Hence task time is

$$Time = q \left[nk(h + p_s 2\psi) + nkp_s \Big(4\delta_1 + (\frac{e_s}{f_s} + \frac{2e_c}{f_c}) \frac{\delta_2 - \delta_1}{l} \Big) \right].$$

- As s becomes large in Case (3) to Case (6), S_i and C cannot fit in L2 nor L3, and they need to be fetched periodically from memory if not L3.

We illustrate s value for the optimal case. For the AMD Bulldozer 8-core CPU architecture (FX-8120) tested in our experiments, L1 cache is of size 16KB for each core. L2 cache is of size 2MB shared by 2 cores and L3 cache is of size 8MB shared by 8 cores. Thus 1MB on average for each core. Other parameters are: $\delta_m = 64.52ns$, $\delta_3 = 24.19ns$,$\delta_2 = 3.23ns$, $\delta_1 = 0.65ns$, $l = 64$ bytes. We estimate $\psi = 0.16ns$, $h = 10ns$, $p_s = 10\%s$, $f_c = f_s = 4$ based on the results from our micro benchmark. The minimum task time occurs in Case (2) when S_i and C can fit in L2 cache, but not L1. Thus the constraint based on the L2 cache size can be expressed as

$$s \times k \times e_s + 2s \times e_c \leq 1MB.$$

While satisfying the above condition, split size s is chosen as large as possible to reduce q value. For Twitter data, k is 18, e_s is 28 bytes, and e_c is 4 bytes. Thus the optimal s is around 2K.

To show how the choice of s affects the task execution time in Formula (1), we measure the ratio of the data access time (including the inverted index lookup) over the computation time:

$$\frac{\text{Data-access}}{\text{Computation}} = \frac{4\delta_1 + (\frac{e_s}{f_s} + \frac{2e_c}{f_c})\frac{\delta_2 - \delta_1}{l} + \frac{h}{p_s}}{2\psi}.$$

This ratio captures the data access overhead paid to perform comparison computation and the smaller the value is, the better. For Twitter benchmark, the above ratio is 8 for optimum case, while it increases to over 25 for Case (3) and Case (4) where more frequent access to L3 cache is required. The data-access-to-computation ratio deduction is supported by experiment results shown in Figure 5. It shows that by selecting s based on our cost function, we are able to reduce the data-access-to-computation ratio from 25 to 8. When an optimum s is chosen, we manage to dramatically reduce the slow cache/memory access out of the whole task execution time.

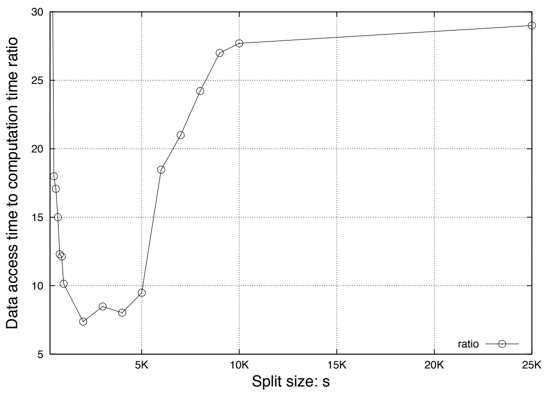


Figure 5: Y axis is the ratio of actual data access time to computation time for Twitter data observed in our experiments.

5. PSS2 WITH FEATURE-BASED VECTOR COALESCING

In PSS1, every time a feature weight from area S_i is loaded to L1 cache, its value is multiplied by a weight from a vector in B. As S_i does not fit in L1 cache, the utilization of L1 for S_i is low. L1 cache usage for S_i is mainly for spatial locality. Namely fetching one or few cache lines for S_i to avoid future L1 cache miss when consecutive data is accessed. The benifit of temporal locality is low, because the same element is unlikely to be accessed again before being evicted, especially for L1 cache due to its small size.

Another way to understand this weakness is that the number of times that an element in L1 loaded for S_i can be used to multiply a weight in B is low before this element of S_i is evicted out from L1 cache. PSS2 is proposed to adjust the data layout and access structure in B in order to increase L1 cache reuse ratio for S_i. The key idea of PSS2 is listed as follows.

- Once an element in S_i is loaded to L1 cache, we compare more vectors in B at each stage. Namely group S_i from S is compared with b vectors in B.

- We coalesce b vectors in B and build an inverted index from these b vectors. The comparison between S_i and b vectors in B is done by intersecting postings of common features in B and S_i.

- The above approach also benefits the amortization of inverted index lookup cost. In PSS1, every term posting lookup for S_i can only benefit multiplication with one element in B. In PSS2, every look up can potentially benefit multiple elements because of vector coalescing.

Figure 7 illustrates the data traversal pattern of PSS2 with $b = 3$. There is one common feature t_3 that appears in both S_i and B. The posting of t_3 in S_i is { $w_{1,3}$,$w_{2,3}$}

```
PSS2 Task $(A, O)$
 1: Read $A$ and divide it into $q$ splits of $s$ vectors each
 2: Build an inverted index for each split $S_i$.
 3: repeat
 4:   Fetch $b$ vectors from $O$ and build inverted index in $B$
 5:   for $S_i \in S$ do
 6:     Compare($S_i$, $B$)
 7: until all vectors in $O$ are compared

Compare $(S, B)$
 8: Initialize array $score$ of size $s \times b$ with zeros
 9: for $j = 1$ to $b$ do
10:   $r_j \leftarrow ||d_j||_1$
11: for Feature $t$ appears in $B$ and $S$ do
12:   for $d_i \in Posting(t)$ in $S$ do
13:     for $d_j \in Posting(t)$ in $B$ and $d_i$ is a candidate do
14:       $score[i][j] = score[i][j] + w_{i,t} \times w_{j,t}$
15:       if $(score[i][j] + maxw[d_i] \times r_j < \tau)$ then
16:         Mark pair $d_i$ and $d_j$ as non-candidate
17:   for $d_j \in Posting(t)$ in $B$ do
18:     $r_j = r_j - w_{j,t}$
19: for $i = 1$ to $s$ do
20:   for $j = 1$ to $b$ do
21:     if $score[i][j] > \tau$ then
22:       Write $(d_i, d_j, score[i][j])$
```

Figure 6: PSS2 task.

and each iteration of PPS2 uses one element from this list, and multiplies it with elements in the corresponding posting of B which is $\{w_{4,3}, w_{6,3}\}$. Thus every L1 cache loading for S_i can benefit 2 multiplications with weights in B. In comparison, every L1 loading of weights for S_i in PSS1 can only benefit one multiplication.

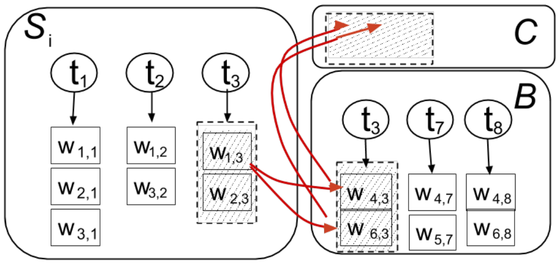


Figure 7: Example of data traversal in PSS2.

Increasing b values expands the size of areas $\mathcal{B}$ and $\mathcal{C}$ to store b vectors and a 2D array $score[][]$. $\mathcal{B}$ and $\mathcal{C}$ may not fit in L1, or even L2 cache anymore. Since L2/L3 cache has higher latency, cache capacity restricts the value of b from being too large. On the other hand, vectors in $\mathcal{B}$ are sparse and b cannot be too small so that there is a sufficient number of vectors sharing a feature after coalescing. Our experiment in Figure 11 discusses this issue in more details.

Similar to PSS1, we can conduct a case-by-case analysis for cache miss ratios of PSS2 based on how S_i, B and $\mathcal{C}$ fit in the different levels of cache. Then we can derive the s and b ranges in each case.

6. EXPERIMENTS

We have implemented PSS, PSS1 and PSS2 in Java. During the evaluation, PSS, PSS1 and PSS2 are applied after data preprocessing. In the default setting, static partitioning [1] is adopted to partition the dataset then, a set of parallel tasks is executed following either PSS2 or PSS1. In another setting (Table 3), LSH [11, 26] is applied first before static partitioning.

We also evaluated another design option we refer to as PSS3. PSS3 follows the previous scientific computing research that views a sparse matrix as a collection of dense small submatrices and employs BLAS3 to perform submatrix multiplication [10, 24, 27]. In this case, we represent the feature vectors in S and B as a set of small submatrices and use a highly optimized BLAS3 library called MTJ [13] for the submatrix multiplication.

The evaluation has the following objectives:

1. Compare PSS1 and PSS2 with the baseline PSS using multiple application datasets. Study how the algorithms behave with different dataset sizes.

2. Evaluate the choice and impact of s value for PSS1 and s and b for PSS2.

3. Illustrate the predicted and observed cache hit ratio. Validate the accuracy of the cost model with respect to the actual execution time.

4. Report the overall parallel performance.

5. Evaluate PSS3 to understand the issues of submatrix multiplication for APSS.

Metrics. We report the running time for different algorithms when the static partitioning is given. Since the number of tasks is fixed, the overall parallel time is proportional to the average task running time. Hence we mainly report the average task running time to evaluate the performance impact of adjusting split size and fetched block size. The cost of self-comparison among vectors within a partition is included when reporting the actual cost. To assess the scalability, we report the overall speedup for the parallel performance, and measure the megaflops number as an additional metric.

6.1 Datasets and experimental setup

The experiments are mainly conducted on a cluster of AMD nodes where each node has 8 cores with 3.1GHz AMD Bulldozer FX8120 and 16GB memory. They run the Hadoop MapReduce environment. In reporting parallel speedup, we have used a bigger cluster of Intel 12-core nodes and each node has dual Intel X5650 six-core processors and 24GB memory. The following five datasets are used.

- Twitter dataset containing 20 million tweets collected from approximately 2 million unique users. The average number of features per vector is 18.32.

- A web dataset containing about 50 million web pages, randomly selected from the Clueweb collection distributed by [3]. The average number of features is 320 per web page.

- Enron email dataset containing 619,446 messages from the Enron corpus, belonging to 158 users with an average of 757 messages per user. The average number of features is 107 per message. The corpus contains large numbers of duplicated emails.

- Yahoo music dataset containing 1,000,990 users rating 624,961 songs to investigate the similarity among songs for music recommendation.
- Google news webpages with over 100K news articles crawled from Google.com. The average number of features per article is 830.

The datasets are preprocessed to follow the TF-IDF weighting after cleaning and stopword filtering [18].

6.2 Comparative studies for execution time and impact of parameters

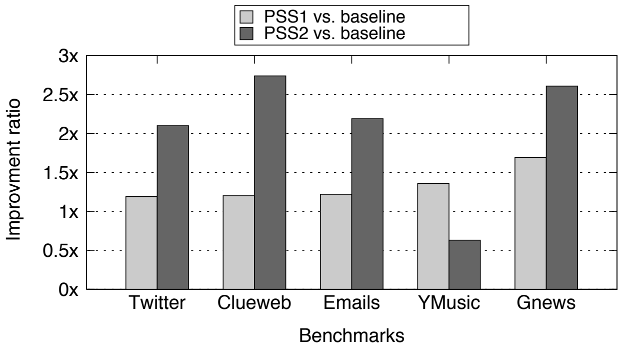


Figure 8: Y axis is ratio $\frac{Time_{PSS}}{Time_{PSS1}}$ and $\frac{Time_{PSS}}{Time_{PSS2}}$. The average task running time includes I/O.

Figure 8 shows the improvement ratio on the average task time after applying PSS1 and PSS2 over the baseline. Namely $\frac{Time_{PSS}}{Time_{PSS1}}$ and $\frac{Time_{PSS}}{Time_{PSS2}}$. PSS is cache-oblivious and each task handles a very large partition that fits into the main memory (but not fast cache). For example, each partition for Clueweb can have around 500,000 web pages. Result shows PSS2 contributes significant improvement compared to PSS1. For example, under Clueweb dataset, PSS1 is 1.2x faster than the baseline PSS while PSS2 is 2.74x faster than PSS. The split size s for PSS1 and s and b for PSS2 are optimally chosen.

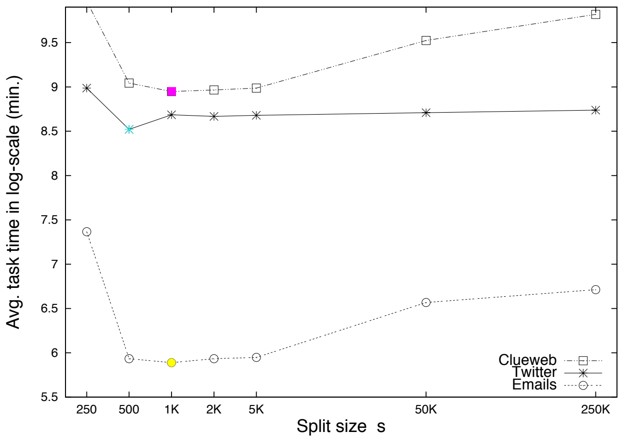


Figure 9: The average running time in *log scale* per PSS1 task under different values for split size s. The partition size S for each task is fixed, $S = s \times q$.

The gain from PSS to PSS1 is achieved by the splitting of the hosted partition data. Figure 9 shows the average running time of a PSS1 task including I/O in log-scale with different values of s. Notice that the partition size ($S = s \times q$) handled by each task is fixed. The choice of split size s makes an impact on data access cost. Increasing s does not change the total number of basic multiplications and additions needed for comparison, but it does change the traversal pattern of memory hierarchy and thus affects data access cost. For all the datasets shown, the lowest value of the running time is achieved when s value is ranged between 0.5K and 2K, consistent with our analytic results.

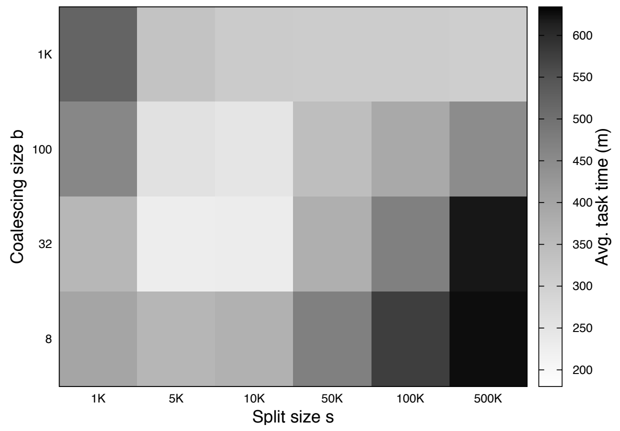


Figure 10: Each square is an $s \times b$ PSS2 implementation (where $\sum s = S$) shaded by its average task time for Twitter dataset. The lowest time is the lightest shade.

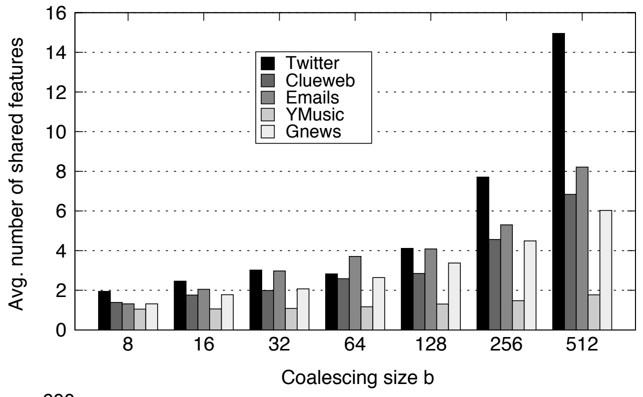
The gain of PSS2 over PSS1 is made by coalescing visits of vectors in B with a control. Figure 10 depicts the average time of the Twitter tasks with different s and b, including I/O. The darker each square is, the longer the execution time is. The shortest running time is achieved when $b = 32$ and s is between 5K to 10K. When b is too small, the number of features shared among b vectors is too small to amortize the cost of coalescing. When b is too big, the footprint of area C and B becomes too big to fit into L2 cache.

While PSS1 outperforms PSS in all 5 datasets, there is an exception for Yahoo music dataset. The benefits of PSS2 over PSS1 depend on how many features are shared in area B. The top and bottom parts of Figure 11 show the average and maximum number of features shared among b vectors in area B, respectively. Sharing pattern is highly skewed and the maximum sharing is fairly high. On the other hand, the average sharing value captures better on the benefits of coalescing. The average number shared exceeds 2 or more for all data when b is above 32 (the optimal b value for PSS2) except Yahoo music. In the Yahoo music data, each vector represents a song and features are the users rating this song. PSS2 slows down the execution due to the low intersection of the interest among users.

6.3 Cache Behavior and Cost Modeling

We demonstrate the cache behavior of PSS1 modeled in Section 4.2 with the Twitter dataset. The Linux perf tool is used to collect the cache miss ratio of L1 and L3.

Figure 12 depicts the real cache miss ratios for L1 and L3 reported by perf tool, the estimated L1 miss ratio which is D_1/D_0, and the estimated L3 miss ratio which is D_3/D_2. L1 cache miss ratio grows from 3.5%, peaks when $s = 8K$, and gradually drops to around 9% afterwards when s value in-



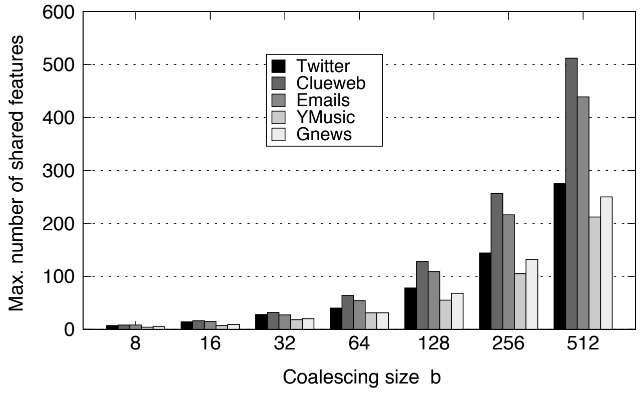


Figure 11: The top is the average number of shared features among b vectors. The bottom is the maximum number of features shared among b vectors.

creases. L3 cache miss ratio starts from 3.65% when s=100, reaches the bottom at 1.04% when s= 5K, and rises to almost 25% when s= 500K. The figure shows that the estimated cache miss ratio approximates the trend of the actual cache miss ratio well.

To validate our cost model, we compare the estimated cost with experimental results in Figure 13. Our estimation of cache miss ratios fits the real ratios quite well, reasonably predicts the trend of ratio change as split size changes. When s is very small, the overhead of building and searching the inverted indexes are too high and thus the actual performance is poor. When s ranges from 50K to 80K, the actual running time drops. This is because as s increases, there is some benefit for amortizing the cost of inverted index lookup. Both the estimated and real time results suggest that the optimum s value is around 2K. Given the optimum s, PSS1 is twice faster than when s is 10K.

6.4 Performance of PSS1 and PSS2 with varying dataset sizes

We also compare the performance of PSS1 and PSS2 with the baseline PSS when the dataset size changes.

Figure 14 shows the average running time of tasks under the three algorithms for four benchmarks with varying input size. We still observe the same trend that PSS1 outperforms the baseline. PSS2 also outperforms PSS1 in all cases except for Yahoo music benchmark. In that case, PSS1 is better than baseline, which is better than PSS2 due to low sharing pattern among the b vectors discussed in Section 6.2.

To reduce the required comparisons with an approximation, we tested the algorithms over an LSH implementation

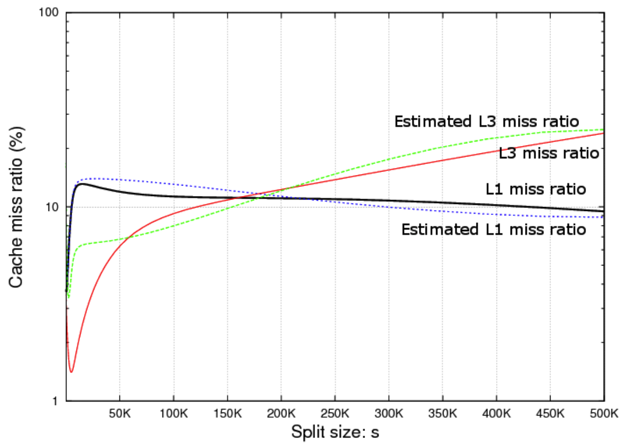


Figure 12: Estimated and real cache miss ratios for PSS1 tasks.

from [26]. LSH is applied first then the static partitioning. Table 3 compares the baseline with PSS2 after applying LSH partitioning over Clueweb dataset with varying sizes. PSS2 is upto 2.55x as fast as PSS for the 50M dataset.

Dataset size	#Buckets	PSS (m)	PSS2 (m)
10 M	176	233.34	98.07
50 M	513	1005.42	394.46

Table 3: PSS and PSS2 task time after LSH mapping for Clueweb. The average number of partitions per bucket is about 6.

6.5 Overall performance and a comparison with PSS3

We assess the overall performance in terms of speedup in processing the entire dataset when varying the number of cores. Figure 15 reports the speedup (parallel time divided by sequential time) for processing Twitter dataset with different numbers of cores in the Intel cluster aforementioned. PSS2 scales well with more computing resources.

We also assess the individual task performance in utilizing the CPU resource by collecting its megaflops rate and compare it with the peak megaflops rate when vectors are dense. Similarity computation can be viewed approximately as a

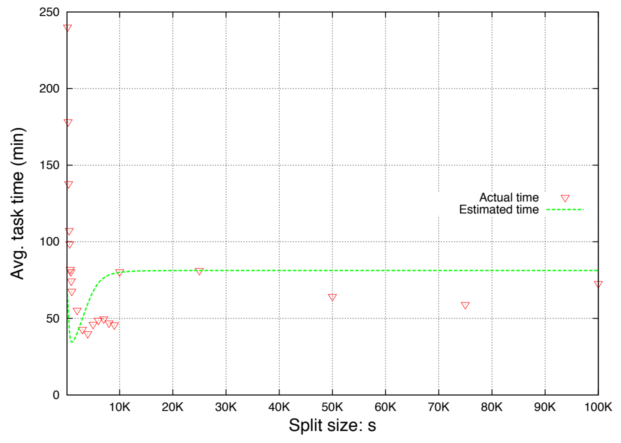
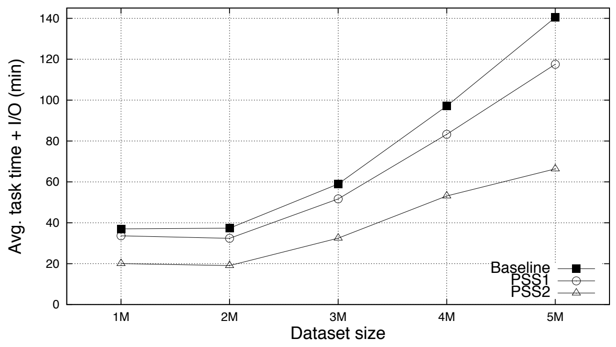
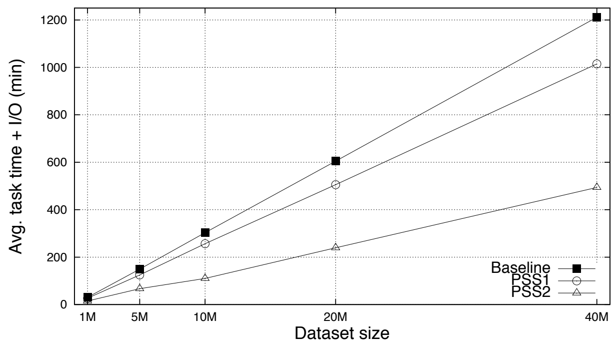


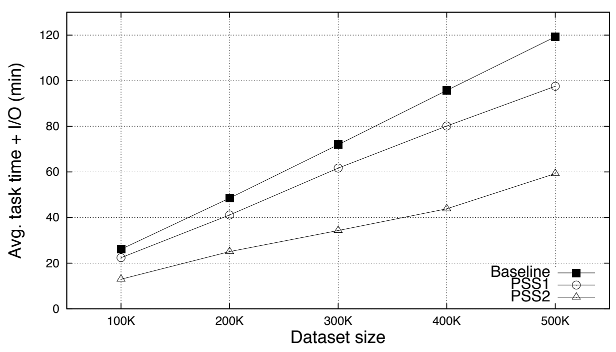
Figure 13: Actual vs estimated average task time for PSS1 in 3M Twitter dataset while split size varies.



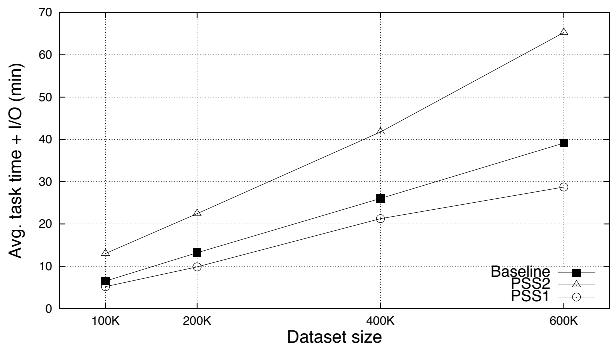
(a) Twitter



(b) Clueweb



(c) Emails



(d) YahooMusic

Figure 14: Average task running time under Baseline, PSS1 and PSS2 over different datasets.

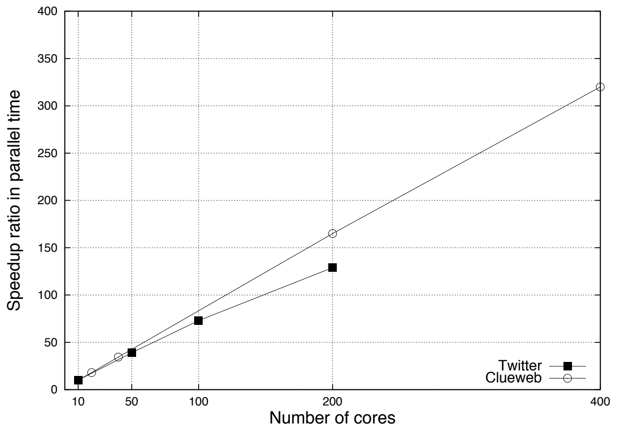


Figure 15: Speedup of PSS2 for processing 20M Twitter and 40M Clueweb with varying numbers of cores.

sparse matrix multiplication together with dynamic computation filtering. We assess the gap between how fast each CPU core can do in terms of peak application performance with a dense matrix and what our scheme has accomplished. First we compare the megaflops performance of our Java code with MTJ [13] from Netlib, which is highly optimized for dense matrix multiplication. The megaflops numbers achieved by a dense matrix multiplication routine (called dgemm) in MTJ achieves 1500 megaflops for matrix dimension 1000 on a single core and achieves 500 megaflops for a small dense matrix. Our scheme achieves 280 megaflops for Twitter benchmark. That is fairly high considering we are dealing with extremely sparse matrices.

In PSS3 design, we represent feature vectors in S and B as a set of small dense submatrices and employ a built-in MTJ BLAS3 dense matrix routine to multiply these submatirces. The advantage of PSS3 is that we leverage MTJ, a highly optimized library for cache performance. The disadvantage is that these small dense matrices still contain many zeros and a BLAS3 routine does not remove the unnecessary computation operations as well as an inverted index does. Figure 16 lists the comparison between PSS3 and PSS2 performance, with the ratio $\frac{Time_{PSS3}}{Time_{PSS2}}$ for different block settings. PSS3 is unfortunately much slower than PSS2. The reason is that vector-feature matrices in the tested similarity applications are extremely sparse and the PSS3 strategy with BLAS3 does not contribute enough benefits to counteract the introduced overhead.

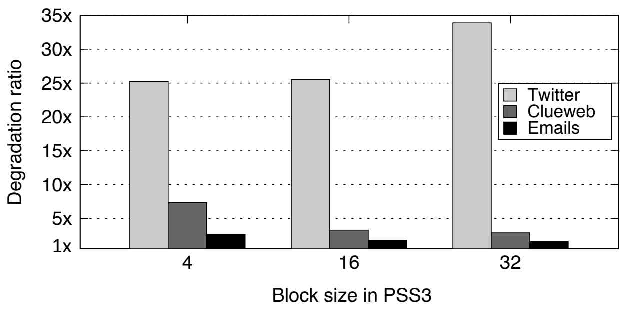


Figure 16: Y axis is ratio $\frac{Time_{PSS3}}{Time_{PSS2}}$. PSS3 is slower than PSS2 in general under different blocking sizes.

Table 4 provides another angle to explain why PSS3 slows down the task. We list the average fill-in ratio of those nonzero submatrices handled by PSS3. Fill-in ratio is the number of stored values which are in fact zero divided by the number of true nonzeros. The fill-in ratio is high and the number of true nonzeros for each block is too low to gain enough benefits with this blocked approach.

Block size	4×4	4×8	4×16	16×16	32×8	32×16
Twitter	2.5	3.7	3.9	6.2	5.3	7.7
Clueweb	2.6	8.2	4.8	5.6	4.4	6.2

Table 4: Average fill-in ratio with different block sizes.

7. CONCLUSIONS

The main contribution of this paper is the development and analysis of cache-conscious data layout and traversal schemes for partition-based similarity search. The key techniques are to 1) split data traversal in the hosted partition so that the size of temporary vectors accessed can be controlled and fit in the fast cache; 2) coalesce vectors with size-controlled inverted indexing so that the temporal locality of data elements visited can be exploited. Our analysis provides a guidance for optimal parameter setting. The evaluation result shows that the optimized code can be upto 2.74x as fast as the original cache-obvious design. Vector coalescing is effective if there is a decent number of features shared among the coalesced vectors.

Acknowledgment

We thank Xifeng Yan, Alexandra Potapova, and Paul Weakliem for their support and feedback, and the anonymous referees for their thorough comments. This work is supported in part by NSF IIS-1118106/0905084 and Kuwait University Scholarship. Equipment access is supported by the Center for Scientific Computing at CNSI/MRL under NSF DMR-1121053 and CNS-0960316. Any opinions, findings, conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

8. REFERENCES

[1] Maha Alabduljalil, Xun Tang, and Tao Yang. Optimizing parallel algorithms for all pairs similarity search. In *Proc. of 6th ACM Inter. Conf. on Web Search and Data Mining (WSDM)*, 2013.

[2] Arvind Arasu, Venkatesh Ganti, and Raghav Kaushik. Efficient exact set-similarity joins. In *VLDB'06*.

[3] Language Technologies Institute at Carnegie Mellon University. The clueweb09 dataset, http://boston.lti.cs.cmu.edu/data/clueweb09.

[4] John R. Gilbert Aydin Bulu. Challenges and advances in parallel sparse matrix-matrix multiplication. In *ICPP*, 2008.

[5] Ricardo Baeza-Yates and Berthier Ribeiro-Neto. *Modern Information Retrieval.* Addison Wesley, 1999.

[6] Roberto J. Bayardo, Yiming Ma, and Ramakrishnan Srikant. Scaling up all pairs similarity search. In *Proceedings of WWW*, 2007.

[7] Fidel Cacheda, Víctor Carneiro, Diego Fernández, and Vreixo Formoso. Comparison of collaborative filtering algorithms: Limitations of current techniques and proposals for scalable, high-performance recommender systems. *ACM Trans. Web*, 2011.

[8] Abdur Chowdhury, Ophir Frieder, David A. Grossman, and M. Catherine McCabe. Collection statistics for fast duplicate document detection. *ACM Trans. Inf. Syst.*, 2002.

[9] J. J. Dongarra, Jeremy Du Croz, Sven Hammarling, and I. S. Duff. A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Softw.*, 16(1):1–17, March 1990.

[10] Iain S. Duff, Michael A. Heroux, and Roldan Pozo. An overview of the sparse basic linear algebra subprograms: The new standard from the blas technical forum. *ACM Trans. Math. Softw.*, 28(2):239–267, June 2002.

[11] Aristides Gionis, Piotr Indyk, and Rajeev Motwani. Similarity search in high dimensions via hashing. In *VLDB*, 1999.

[12] Hannaneh Hajishirzi, Wen tau Yih, and Aleksander Kolcz. Adaptive near-duplicate detection via similarity learning. In *SIGIR*, 2010.

[13] Heimsund Halliday. http://code.google.com/p/matrix-toolkits-java.

[14] Nitin Jindal and Bing Liu. Opinion spam and analysis. In *Proceedings of the international conference on Web search and web data mining*, WSDM '08, pages 219–230, 2008.

[15] David Kanter. Md's bulldozer microarchitecture. *realworldtech.com*, 2010.

[16] Aleksander Kolcz, Abdur Chowdhury, and Joshua Alspector. Improved robustness of signature-based near-replica detection via lexicon randomization. In *Proceedings of KDD*, 2004.

[17] David Levinthal. Performance analysis guide for intel core i7 processor and intel xeon 5500 processors. *Intel*, 2009.

[18] Jimmy Lin. Brute force and indexed approaches to pairwise document similarity comparisons with mapreduce. In *SIGIR*, 2009.

[19] Stefan Manegold, Peter Boncz, and Martin L. Kersten. Generic database cost models for hierarchical memory systems. In *VLDB '02*, 2002.

[20] 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.

[21] Gianmarco De Francisci Morales, Claudio Lucchese, and Ranieri Baraglia. Scaling out all pairs similarity search with mapreduce. In *8th Workshop on LargeScale Distributed Systems for Information Retrieval (2010)*, 2010.

[22] Mehran Sahami and Timothy D. Heilman. A web-based kernel function for measuring the similarity of short text snippets. In *WWW '06*, pages 377–386, 2006.

[23] Ambuj Shatdal, Chander Kant, and Jeffrey F. Naughton. Cache conscious algorithms for relational query processing. In *In Proceedings of the 20th VLDB Conference*, pages 510–521. Morgan Kaufmann Publishers Inc, 1994.

[24] Kai Shen, Tao Yang, and Xiangmin Jiao. S+: Efficient 2d sparse lu factorization on parallel machines. *SIAM J. Matrix Anal. Appl.*, 22(1):282–305, April 2000.

[25] Narayanan Shivakumar and Hector Garcia-Molina. Building a scalable and accurate copy detection mechanism. In *DL'96 (ACM Inter. Conf. on Digital libraries)*, pages 160–168.

[26] Ferhan Ture, Tamer Elsayed, and Jimmy Lin. No free lunch: brute force vs. locality-sensitive hashing for cross-lingual pairwise similarity. In *SIGIR '2011.*

[27] Richard Vuduc, James W. Demmel, Katherine A. Yelick, Shoaib Kamil, Rajesh Nishtala, and Benjamin Lee. Performance optimizations and bounds for sparse matrix-vector multiply. In *ACM/IEEE Conf. on Supercomputing*, 2002.

[28] Chuan Xiao, Wei Wang, Xuemin Lin, and Jeffrey Xu Yu. Efficient similarity joins for near duplicate detection. In *Proceeding of the 17th international conference on World Wide Web*, WWW '08, pages 131–140. ACM, 2008.

[29] Yuan Cao Zhang, Diarmuid Ó Séaghdha, Daniele Quercia, and Tamas Jambor. Auralist: introducing serendipity into music recommendation. In *Proceedings of the fifth ACM international conference on Web search and data mining*, WSDM '12. ACM, 2012.

[30] Shanzhong Zhu, Alexandra Potapova, Maha Alabduljalil, Xin Liu, and Tao Yang. Clustering and load balancing optimization for redundant content removal. In *WWW '12: Inter. Conf. on World Wide Web. Industry Track*, 2012.