Data Filtering for Scalable High-dimensional k-NN Search on Multicore Systems

Xiaoxin Tang[1], Steven Mills[2], David Eyers[2], Kai-Cheung Leung[2],
Zhiyi Huang[2], Minyi Guo[1]
[1]Shanghai Key Laboratory of Scalable Computing and Systems, Shanghai Jiao Tong University, China
[2]Department of Computer Science, University of Otago, New Zealand

ABSTRACT

K Nearest Neighbors (k-NN) search is a widely used category of algorithms with applications in domains such as computer vision and machine learning. With the rapidly increasing amount of data available, and their high dimensionality, k-NN algorithms scale poorly on multicore systems because they hit a memory wall. In this paper, we propose a novel data filtering strategy, named Subspace Clustering for Filtering (SCF), for k-NN search algorithms on multicore platforms. By excluding unlikely features in k-NN search, this strategy can reduce memory footprint as well as computation. Experimental results on four k-NN algorithms show that SCF can improve their performance on two modern multicore platforms with insignificant loss of search precision.

Categories and Subject Descriptors

I.0 [**Computing Methodologies**]: GENERAL

General Terms

Performance

Keywords

K Nearest Neighbors; High-Dimensional Space; Memory Wall; Multicore Systems; Subspace Clustering for Filtering.

1. INTRODUCTION

Similarity search is one of the applications that demands efficient parallel algorithms on multicore systems. Through finding similar items within a known database, existing knowledge can be used for predicting unknown information. Many domains, such as computer vision [14], bioinformatics [3], data analysis [5], handwriting recognition [16], and many other statistical classification tasks, rely on similarity search and demand high-performance algorithms, especially under the pressure of *big data* [10]. For example, the large amount of available images makes image-matching [13] from computer vision a very interesting and challenging problem.

K Nearest Neighbors (k-NN) search is one frequently used category of algorithms for solving similarity search problems. Here, we take the concept "feature" to represent one data item in the database. In general, a feature f can be defined as a D dimensional vector—we later refer to its components as e_1 through e_D. The database X is defined as a set of N such features: $X = \{f_1, f_2, \ldots, f_N\}$. The similarity is often measured by Euclidean Distance (ED). Based on these definitions, the k-NN problem can be formally described as: given a query feature q, find k reference features in X that have the shortest (Euclidean) distances to q.

In general, most algorithms need two types of data structures: index data and feature data, both of which are frequently visited during k-NN search. The index structure is used for finding reference features—called candidate features—that are most likely to be the k nearest neighbors. To decide whether a candidate feature is one of the k nearest neighbors, the feature data will be visited in order to evaluate their similarity. The feature data structure is a matrix and can consume up to $O(ND)$ memory space.

As image-matching applications are becoming more and more popular, the size of typical feature sets X is increasing. The dimensionality of features is also high: e.g. SIFT [9] features have 128 dimensions. When both N and D are very large, which is often the case of problems like image matching, the feature structure can consume up to several dozens of megabytes for a single image. In this case, many available algorithms do not work efficiently on multicore systems [13] due to memory latency and bandwidth limitations (also known as the *memory wall*), as the data structure is not small enough to fit in the last-level cache.

In this paper, we propose a novel *data filtering* strategy for high-dimensional k-NN search on multicore systems. Instead of finding the likely candidates, our data filtering strategy excludes those unlikely features based on distance estimation. The data filtering strategy has two advantages. First, it reduces computation and the number of memory accesses by replacing high-dimensional distance calculation with simple distance estimation. Second, its index structure for filtering has a very small memory footprint and thus reduces the effect of memory wall.

This paper is organized as follows: Section 2 presents the SCF method. Section 3 shows performance results of SCF that is applied to four k-NN algorithms on multicore systems. Section 4 discusses the related work. Finally, Section 5 draws conclusions of this paper.

2. THE DATA FILTERING STRATEGY

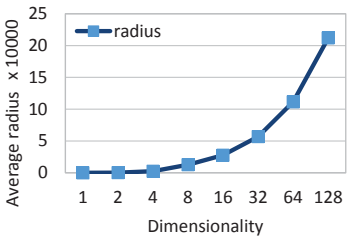
In this section, the following Squared Euclidean Distance (SED) is used to measure the similarity between two features:

$$SED(f_i, f_j) = \|f_i - f_j\| = \sum_{m=1}^{D} (f_i[m] - f_j[m])^2. \quad (1)$$

The square root in ED is not used in the SED, which can reduce the computation without changing the search results.

HPDC'14, June 23–27, 2014, Vancouver, BC, Canada.

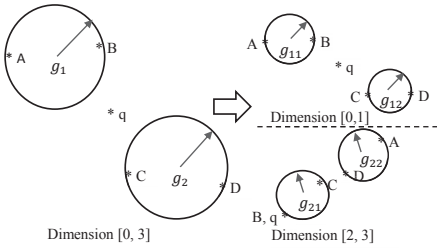
http://dx.doi.org/10.1145/2600212.2600710.



(a) Average radius of a random dataset.

	e_1	e_2	e_3	e_4
q	0	0	0	0
A	-4	2	3	4
B	-2	2	1	0
C	2	-3	0	1
D	4	-3	4	3
g_1	-3	2	2	2
g_2	3	-3	2	2

(b) A 4-dimensional case.



(c) From a full-space clustering to subspace clustering.

Figure 1: Challenges of using clustering for distance estimation in high-dimensional space. (a) The left figure shows the average radius of a randomly generated dataset. This dataset contains 10,000 features, which are divided into 32 groups. Each element of the features is uniformly distributed in the range of [1, 128]. (b) The table in the middle gives a simple 4-dimensional example. (c) The right figure shows our subspace clustering method.

2.1 A case study: brute-force search

Here we use the brute-force k-NN search to demonstrate how our data filtering strategy works. To find the k-NN of a given query feature, brute-force search first calculates all the distances between the query feature and all reference features in the database. It uses a max-heap of size k to accumulate the features with the smallest distances. After all of the distances are pushed onto the heap, the k-NN results can be collected from it. This algorithm is very computation-intensive as it will cost[1] $O(ND)$ to calculate the distances and $O(N \log k)$ to find the k-NNs. Distance calculation will dominate the time as $\log k$ is very small for small k while D can be large for high-dimensional problems. It also has a large memory footprint as it needs to scan the whole database for each query.

Since k is usually much smaller than the size of the database X, many distance calculations are not necessary as most features are far away from the query feature. If we can exclude those features that are unlikely to be a k-NN using simple distance estimation, we can reduce the computation as well as the memory footprint.

2.2 Distance estimation through clustering

The key issue now is how to estimate the distances accurately and efficiently. Clustering is a traditional method that is used to estimate the distances to a group of features. In this paper, we use the k-means algorithm of the FLANN library [11] for subspace clustering in our distance estimation. Though better clustering methods may be used, they do not affect our general approach.

After clustering, each reference feature will be assigned to the group whose group center is the closest to that reference feature. Then, these group centers will represent the features within their corresponding groups. However, when the dimensionality becomes large, the features are sparsely distributed in the space and the radius of each group becomes large as well. For example, Figure 1a gives the average radius of the groups generated from random dataset with variable dimensionality. As we can see, the radius of the groups grows quickly with the increasing dimensionality. When the radius is large, clustering-based distance estimation becomes less accurate.

Consider a simple 4-dimensional case as an example, which is given in Figure 1b. Here, q is the query feature; A, B, C and D are four reference features. After clustering on the reference features based on the all four dimensions, A and B are put into the same group with the center g_1, and C and D are put into the other group with the center g_2. The left side of Figure 1c illustrates the clustering result (it is simplified with circles as it is hard to draw 4-dimensional space). If we use this clustering result to estimate distances between the query and the reference features, then $\|g_1 - q\|$ will represent $\|A-q\|$ and $\|B-q\|$ while $\|g_2-q\|$ will represent $\|C - q\|$ and $\|D - q\|$. As $\|g_1 - q\| = 21$ and $\|g_2 - q\| = 26$, the order of the reference features based on the distance estimation is A, B, C, D. However, their real distances are $\|A - q\| = 52$, $\|B-q\| = 8$, $\|C-q\| = 15$ and $\|D-q\| = 43$, and the right order should be B, C, D, A. If $k = 1$, the results based on this distance estimation will have 0% accuracy, while in the case of $k = 2$, the accuracy is only 50%.

From the above example we can see that clustering within high-dimensional spaces has two problems. First, it is so coarse-grained that it is not able to tell the differences between features within the same group. For example, it cannot tell that B is much closer to q than A. Second, it could present incorrect results easily as a closer group center does not mean all features in that group are closer to the query. For example, though group g_1 is closer to q than group g_2, feature C in g_2 has a smaller distance to q than A of group g_1. The reason is that the radius of each group could be very large, and thus can obscure the differences between groups.

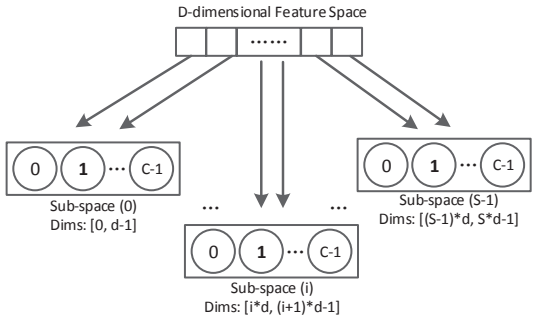


Figure 2: The basic structure for SCF method. It contains S subspaces. All features are divided into different groups by using the corresponding dimensions within each subspace.

2.3 Subspace Clustering for Filtering

Based on the above analysis, we propose the following Subspace Clustering for Filtering (SCF) method. As Figure 2 shows, the data structure of SCF is a multi-level cover of the feature space. Instead of using all the dimensions for clustering, SCF divides the whole space into S subspaces, each of which may contain $\lfloor \frac{D}{S} \rfloor$ dimen-

[1]Big-O notation usually denotes asymptotic effects, but we will use it as shorthand for proportionality without simplified expressions.

Algorithm 1: Build the SCF index.

$d \leftarrow \lfloor \frac{D}{S} \rfloor$;
for $i \leftarrow 0$ **to** $S-1$ **do**
  Based on dimensions $[i \times d, (i+1) \times d)$, use a clustering method (e.g. k-means) to divide X into C groups;
  for $j \leftarrow 0$ **to** $N-1$ **do**
    $\beta[j][i] \leftarrow$ group ID that feature j belongs to;
  for $j \leftarrow 0$ **to** $C-1$ **do**
    $\theta[i][j] \leftarrow$ center of group j;
    $\gamma[i][j] \leftarrow$ radius of group j;
return β, θ and γ;

Table 1: PSEDs between q and the group centers in the example.

	g_{11}	g_{12}	g_{21}	g_{22}
q	13	18	0.5	24.5

sions. The remainder of $\frac{D}{S}$ can either be treated as an additional subspace, or these dimensions can be distributed to the other subspaces. Then, within each subspace, we use the aforementioned k-means clustering method to divide the features into C different groups where each group may contain $\frac{N}{C}$ features on average.

The SCF-based distance estimation depends on two data structures: the SCF index and a matrix of partial distances for the query feature. The SCF index is created based on the clustering results in the subspaces. The detailed algorithm for creating SCF index is shown in Algorithm 1. β, θ and γ in the algorithm are three matrixes that represent the SCF index. Each element β_{ij} ($i \in [0, N)$, $j \in [0, S)$) in the index represents the group ID of the i^{th} feature of X within the j^{th} subspace. θ_{jt} ($t \in [0, C)$) represents the center point of the t^{th} group in the j^{th} subspace. Similarly, γ_{jt} is used to represent the radius of the t^{th} group in the j^{th} subspace.

The matrix of partial distances for the query feature is created by Algorithm 2. It is represented by the matrix δ in the algorithm. The matrix gives the Partial SED (PSED) between the query feature and the center of each group in each subspace. It can be defined as:

$$PSED_{l,u}(f_i, f_j) = \sum_{m=l}^{u} (f_i[m] - f_j[m])^2 \qquad (2)$$

where $1 \le l \le u \le D$, and $[l, u]$ bound the dimensions used to form a subspace.

Algorithm 3 shows the steps for distance estimation. The PSED between the query and the center of a group is used to estimate the PSED between the query and the reference features of that group. For each reference feature, the sum of all estimated PSEDs in every subspace is used as the Estimated SED (ESED) between the query and the reference feature.

Table (1) shows the matrix for the PSEDs of the previous example, where g_{11}= $(-3, 2, \cdot, \cdot)$, g_{12}= $(3, -3, \cdot, \cdot)$, g_{21}= $(\cdot, \cdot, 0.5, 0.5)$, and g_{22}= $(\cdot, \cdot, 3.5, 3.5)$. Thus, in the right side of Figure 1c, the ESED of each reference features are:

$$\|A-q\|_{\text{est}} = \|g_{11}-q\|_{\text{psed}} + \|g_{22}-q\|_{\text{psed}} = 37.5,$$
$$\|B-q\|_{\text{est}} = \|g_{11}-q\|_{\text{psed}} + \|g_{21}-q\|_{\text{psed}} = 13.5,$$
$$\|C-q\|_{\text{est}} = \|g_{12}-q\|_{\text{psed}} + \|g_{21}-q\|_{\text{psed}} = 18.5,$$
$$\|D-q\|_{\text{est}} = \|g_{12}-q\|_{\text{psed}} + \|g_{22}-q\|_{\text{psed}} = 42.5.$$

They result in the estimated order B, C, A, D, which is closer to the real order of B, C, D, A than that estimated based on the original full-space clustering.

Algorithm 2: Calculation of partial distances between the query feature and the center of each group in each subspace

$d \leftarrow \lfloor \frac{D}{S} \rfloor$;
$\delta[S][C] \leftarrow 0$;
for $i \leftarrow 0$ **to** $S-1$ **do**
  for $j \leftarrow 0$ **to** $C-1$ **do**
    $l \leftarrow i \times d$;
    $u \leftarrow (i+1) \times d - 1$;
    $\delta[i][j] \leftarrow PSED_{[l,u]}(q, \theta[i][j])$
return δ;

Algorithm 3: $SCF\_Estimation(q, r_t)$

$ESED \leftarrow 0$;
for $i \leftarrow 0$ **to** $S-1$ **do**
  $ESED \leftarrow ESED + \delta[i][\beta[t][i]]$;
return $ESED$

It is worth noting that the overhead of Algorithm 1 is a one-off cost, which will be relatively minor when amortized over many queries. Also note that by adjusting S and C in the above algorithms, we can change the estimation accuracy of SCF. Usually when S and C are increasing, the estimation accuracy can be improved. Since this paper focuses on performance and due to the limited space here, we do not give further discussions on how to maintain a high estimation accuracy. However, the real accuracy achieved by our method is given in the experimental section.

2.4 Space complexity analyses

As shown in the above algorithms, SCF uses small index structures. Since there are S subspaces and each one has C groups, it takes $O(SC\frac{D}{S}) = O(CD)$ memory space to store all the group centers (θ) and $O(SC)$ memory space to store radius of each group (γ). Then, it takes $O(NS)$ memory space to store group IDs (β) for all reference features. During runtime, it will cost $O(SC)$ memory space to store the PSEDs (δ) for each query feature. Overall, the total memory used is $O(CD+SC+NS+SC)$ for SCF method. As N is the dominant one among all parameters, the space complexity for SCF can be reduced to $O(NS)$. Since S is much smaller than D (8 versus 128 in our implementation for SIFT dataset), the index structure of SCF is more likely to fit into the shared cache. For example, when $N = 20000$, the brute-force algorithm needs to access up to 10 MiB memory (each element of the feature is float number) while the SCF structure only needs around 160 KiB (group ID is represented by one byte). Therefore, SCF can better utilize the shared cache and requires significantly fewer memory accesses compared to the brute-force algorithm.

3. EVALUATION

In this section, we evaluate the performance of our SCF method when it is applied to four k-NN algorithms: Brute-force (BF), Randomized KD-Trees (RKD), Hierarchical k-means (Kmeans) and Random Ball Cover (RBC). The first three algorithms (BF, RKD and Kmeans) are chosen from the FLANN [11] library, which is also contained in OpenCV [2] to provide fast approximate k-NN search functionality. RBC is a state-of-the-art algorithm on parallel platforms [4] and is well optimized to reduce scalability problems when running on multicore systems. As the BF algorithm is the most computation- and memory-intensive algorithm, we use it to show that the SCF method can effectively reduce computation and

Table 2: Filtering rate (FR) and lost precision (LP) after applying SCF on each algorithm and dataset.

Algs \ Dataset	SIFT		Random		Madelon		HAR		Digits	
	FR	LP	FR	LP	FR	LP	FR	LP	FR	SP
BF_SCF	96.87%	3.23%	89.53%	3.76%	67.75%	0.58%	89.21%	0%	96.64%	3.51%
RKD_SCF	82.99%	2.83%	84.48%	3.54%	20.22%	0.08%	76.88%	3.81%	66.84%	3.05%
Kmeans_SCF	77.66%	3.49%	87.43%	2.96%	48.39%	0%	34.5%	3.38%	47.38%	3.64%
RBC_SCF	87.81%	2.72%	85.75%	2.14%	48.87%	0%	68.38%	3.97%	81.24%	4.61%

Table 3: Overview of each test dataset.

Name	Ref	Query	Dim
SIFT	25271	7481	128
Random	25000	7500	128
Madelon	2000	1800	500
HAR	7352	2947	560
Digits	3823	1797	64

memory footprint. However, we will also demonstrate that the filtering method is very effective when applied to other optimized algorithms such as RBC.

The datasets listed in Table (3) are used to evaluate the performance of the above algorithms. In the table, "SIFT" represents features generated by the SIFT [9] algorithm, which is commonly used in computer vision. "Random" contains features that are randomly generated and evenly distributed in the feature space (a hypercube with sides of length 128). The "Digits", "Madelon" and "HAR" datasets are selected from the UCI Machine Learning Repository [1]. In Table (3), the "Ref" column indicates the number of reference features while the "Query" column lists the number of query features used in the experiment. The "Dim" column specifies the dimensionality of the datasets.

Two multicore platforms are used in our evaluation:

- AMD64: AMD Opteron Processor 6276, 16 cores $\times$ 4 @ 2.3 GHz, 16 MiB L3 shared cache, 64GiB DDR3 (1333 MHz) memory;
- MIC: Intel Xeon Phi Coprocessor 5110P, 60 cores @ 1.0 GHz, 30 MiB L2 shared cache, 8 GiB GDDR5 (5.5 GHz) memory.

The g++-4.4 compiler is used on the AMD64 machine and icc-14.0 is used for the code generation for the Xeon Phi.

3.1 Performance of sequential execution

In this section, we evaluate the performance after applying SCF to the aforementioned four algorithms under sequential execution. The results are collected from running the algorithms on a single core of AMD64. As shown in Table (2), two metrics are used to evaluate the performance and precision of SCF. The first one is Filtering Rate (FR), which represents the percentage of features that can be filtered by SCF. Thus, the higher the FR, the more computation and memory accesses it reduces, which leads to better performance. The second one LP, indicates the lost precision compared with the original k-NN results. For example, the LP of RBC_SCF is the number of k-NN that are not in the k-NN results of the original RBC, divided by the total number of k-NN of the original RBC in each test. From the table we can see that SCF can successfully maintain a LP of under 5%.

Though LP is very small in Table (2), FR varies across different datasets and algorithms. This is because different algorithms have different search precisions on different datasets. For example, RKD can find the k-NN of "Madelon" efficiently, which leads to a lower FR (20.22%). In this case most features the original RKD has found are good candidates that SCF cannot exclude. Similarly, Kmeans processes "HAR" well, and thus SCF achieves a lower FR (34.5%). SCF works well on BF and RBC in most cases as both algorithms are highly dependent on exhaustive search of the feature space, which is very suitable for applying SCF.

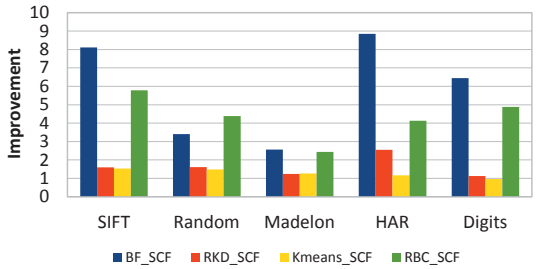


Figure 3: Performance improvement of sequential execution after applying SCF to each algorithm on AMD64 machine.

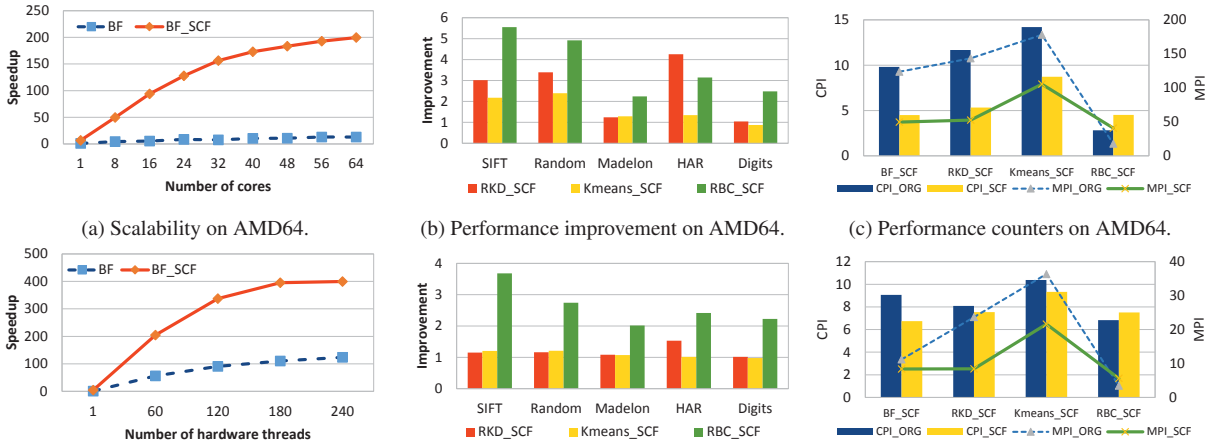
Figure 3 gives the performance improvement on a single core of AMD64 after applying SCF to each algorithm. As we can see, SCF can improve the performance by up to $8.85\times$ for BF (in the "HAR" case) and up to $5.78\times$ for RBC ("SIFT"). This can be explained by the exhaustive search in both algorithms benefiting greatly from SCF. Though FR for RKD and Kmeans is high for some datasets, their performance improvement is not as good as BF and RBC. This is because both RKD and Kmeans spend a lot of time searching their complex index structures to get a small number of good candidates. Since the number of candidates for filtering is small, SCF has a smaller effect on these two algorithms, even though FR is high. However, on average, SCF can still improve the performance of RKD by 33% and that of Kmeans by 19%. Moreover, on multicore platforms, RKD and Kmeans will benefit more from SCF due to reduced memory accesses, as we demonstrate later.

3.2 Performance of parallel execution

Although the computing power is increasing on multicore machines, memory latency and bandwidth are often the bottleneck that leads to poor performance. We will show that, after applying our SCF method, the scalability of the k-NN algorithms on multicore machines is greatly improved. Here, all algorithms are parallelized by using OpenMP and the suffix "_SCF" means that SCF is applied to the corresponding algorithm. The improvements are calculated by comparing with the original algorithm. For example, the improvement for BF is calculated as the execution time of the parallelized original BF divided by the time of the parallelized BF_SCF.

Table 4: Parallel performance improvement of BF_SCF over the original BF algorithm on each platform and dataset.

Platform	SIFT	Random	Madelon	HAR	Digits
AMD64	15.54×	5.04×	2.66×	9.43×	4.13×
MIC	3.23×	2.11×	1.43×	2.97×	1.33×



(a) Scalability on AMD64.

(b) Performance improvement on AMD64.

(c) Performance counters on AMD64.

(d) Scalability on MIC.

(e) Performance improvement on MIC.

(f) Performance counters on MIC.

Figure 4: Performance statistics of SCF.

3.2.1 *Performance improvement of the BF_SCF*

Table (4) lists the parallel performance improvement of BF_SCF on AMD64 and the MIC machines. Compared with their sequential performance shown in Figure 3, the BF_SCF search has the most improvement. For the case of the SIFT dataset on AMD64, its improvement is $15.54\times$ (64 cores), which is much better than the $8.11\times$ on a single core. Figure 4a explains why the parallel BF_SCF is able to get more performance gain than its sequential counterpart. The speedup curves in the figure show the good scalability of BF_SCF, while the original BF's speedup curves become flat after 32 cores. On the AMD64 machine, the BF hits the memory wall much earlier than when all cores are used.

This result shows that for an embarrassingly parallel algorithm like BF, the memory wall becomes one of the most serious bottlenecks, which is supported by our statistics collected from performance monitoring counters. However, after applying SCF, its scalability has been significantly improved. For example, the speedup against the original sequential BF has been improved from $12.84\times$ to $199.63\times$ on AMD64 when all cores available are used. On the MIC platform, the scalability of the original BF is better because MIC has much better memory bandwidth. Moreover, since MIC has four hardware threads in each core, it can efficiently hide the memory latency through overlapping computation and memory access. In this case, the memory wall problem in the original BF is greatly relieved and it has reasonable scalability on MIC, as shown in Figure 4d. However, BF_SCF still has much better performance than the original BF, as can be seen in the other series on that figure.

3.2.2 *Performance improvement of other algorithms*

Figures 4b and 4e show the performance improvement of other k-NN algorithms on parallel platforms. This compares the original algorithm running across all cores to the SCF version. The performance improvement of RBC_SCF is very similar to that of its sequential counterpart ($5.64\times$ versus $5.54\times$ on AMD64 in the best cases). Since this algorithm has already been optimized for multicore platforms, it scales well on parallel platforms and does not suffer from the memory wall. This shows that SCF is very cache-efficient and has little impact on the performance of those algorithms that already have good cache utilization. On AMD64, RKD_SCF and Kmeans_SCF get their best performance improvement of $4.25\times$ and $2.39\times$, which is much better than their sequential improvement ($2.55\times$ and $1.53\times$).

However, for the "Madelon" and "Digits" datasets, neither the RKD_SCF nor Kmeans_SCF algorithms have more of a performance improvement than their sequential counterparts do. The reason is that both datasets are quite small (3.8 MiB for "Madelon" and 0.88 MiB for "Digits") so that they can fit in the last-level cache and are less likely to hit the memory wall. Moreover, due to the lower dimensionality, RKD and Kmeans perform efficiently on "Digits" anyway. Thus, fewer features can be filtered by SCF. Nonetheless, in most cases SCF can significantly improve performance in these algorithms on AMD64.

Since MIC has a higher memory bandwidth, the memory wall problem is relieved for the k-NN algorithms. This is due to its usage of the GDDR5 memory and a larger shared L2 cache that provides very high memory throughput. The performance improvement of most algorithms after applying SCF is quite similar to their sequential counterparts, which means they scale well on this new platform.

We note that the current evaluation code does not contain low-level optimizations specific to the architecture, and thus its computing ability may not be fully utilized. For example, the Vector Processing Unit (VPU) in Xeon Phi contributes most to the platform's peak computing power. If the VPUs are fully utilized, the memory latency may again become the bottleneck. We will explore this in our future work.

3.2.3 *Performance monitoring counter statistics*

Figure 4c and 4f are provided to verify our previous observations and analyses. In the figures, Cycles Per Instruction (CPI) is used to evaluate the computing efficiency while Misses Per Instruction (MPI) is used to represent intensity of the last-level cache misses per instruction. For AMD64, the CPIs have a very close relationship with the MPIs as they grow and drop in the same pattern. That means that the CPIs are mainly affected by the memory wall. However, for MIC, CPI is not significantly influenced by MPI, which demonstrates that the Xeon Phi can provide enough memory bandwidth for these algorithms.

In summary, SCF is general enough to improve the performance of existing k-NN algorithms on different datasets by reducing both computation and memory accesses. Both memory-intensive and computation-intensive k-NN algorithms can benefit from our proposed method.

4. RELATED WORK

As far as we know, this is the first effort on optimization of approximate k-NN algorithms on multicore systems that addresses both performance and precision.

Garcia *et al.* [6] first used the GPU to implement the brute-force algorithm. However, as implementing efficient max-heaps on GPU is very difficult, it becomes very slow in searching for the smallest distances, especially when the required number of results (k-NN) is larger than 2 [13]. Designing other multicore-friendly approximate algorithms has been a recent trend for accelerating k-NN search (e.g. RBC [4]). Although they have achieved very good performance on multicore platforms, they still incur a great deal of unnecessary computation, which can be reduced with our data filtering mechanism.

The Vector Approximation (VA) [15] and Vector Quantization (VQ) [12] approaches share a similar idea of using small structures to represent data and estimate distances. However, they are designed to reduce disk I/O overhead. While VA uses one dimension and VQ uses full dimensions to build the index, our method can choose any number of dimensions to better balance time complexity and estimation accuracy. Location Sensitive Hashing (LSH) [3] uses special hash functions so that features that are close to each other will get the same hash value. However, developing an appropriate hash function can be a very complex undertaking [4].

The Xeon Phi is a new coprocessor with the Intel Many Integrated Core (MIC) architecture. Currently, many researchers are exploring this new architecture. For example, Alexander *et al.* have implemented the famous Linpack Benchmark on Xeon Phi [7], and Liu *et al.* have designed efficient sparse matrix-vector multiplication on this new architecture [8]. As far as we know, our work is the first effort evaluating the performance of k-NN algorithms on Xeon Phi.

5. CONCLUSIONS

Traditional k-NN algorithms run into serious bottlenecks caused by the memory wall on multicore systems. In this paper, we propose a data filtering strategy that tries to reduce the computation- and memory-intensive distance calculation. We propose the Subspace Clustering for Filtering (SCF) method, which can accurately estimate similarity. Experimental results show that SCF is general enough to significantly improve the performance of several k-NN algorithms on multicore platforms.

In the future, we intend to further explore how to improve our method so that it can efficiently utilize the massive computing ability and memory bandwidth of new hardware such as next generation GPUs and the Xeon Phi.

Acknowledgment

We thank the anonymous reviewers for their valuable comments. Xiaoxin Tang would like to thank the University of Otago for hosting his PhD internship during the course of this research. This work was partially supported by the Program for Changjiang Scholars and Innovative Research Team in University (IRT1158, PCSIRT) China, NSFC (Grant No. 61272099, 61261160502) and by the Scientific Innovation Act of STCSM (No. 13511504200).

6. REFERENCES

[1] K. Bache and M. Lichman. UCI machine learning repository, 2013.

[2] G. Bradski and A. Kaehler. *Learning OpenCV: Computer vision with the OpenCV library*. O'Reilly Media, Incorporated, 2008.

[3] J. Buhler. Efficient large-scale sequence comparison by locality-sensitive hashing. *Bioinformatics*, 17(5):419–428, 2001.

[4] L. Cayton. Accelerating nearest neighbour search on manycore systems. In *IEEE Int. Parallel and Distributed Processing Symposium (IPDPS)*, 2012.

[5] D. L. Donoho et al. High-dimensional data analysis: The curses and blessings of dimensionality. *AMS Math Challenges Lecture*, pages 1–32, 2000.

[6] V. Garcia, E. Debreuve, F. Nielsen, and M. Barlaud. K-nearest neighbor search: Fast GPU-based implementations and application to high-dimensional feature matching. In *Image Processing (ICIP), 2010 17th IEEE International Conference on*, pages 3757–3760, 2010.

[7] A. Heinecke, K. Vaidyanathan, M. Smelyanskiy, A. Kobotov, R. Dubtsov, G. Henry, A. G. Shet, G. Chrysos, and P. Dubey. Design and implementation of the Linpack benchmark for single and multi-node systems based on Intel Xeon Phi coprocessor. *Parallel and Distributed Processing Symposium, International*, 0:126–137, 2013.

[8] X. Liu, M. Smelyanskiy, E. Chow, and P. Dubey. Efficient sparse matrix-vector multiplication on x86-based many-core processors. In *Proceedings of the 27th international ACM conference on International conference on supercomputing*, ICS '13, pages 273–282, New York, NY, USA, 2013. ACM.

[9] D. Lowe. Object recognition from local scale-invariant features. In *Computer Vision The Proceedings of the Seventh IEEE International Conference on*, volume 2, pages 1150–1157 vol.2, 1999.

[10] J. Manyika, M. Chui, B. Brown, J. Bughin, R. Dobbs, C. Roxburgh, and A. H. Byers. Big data: The next frontier for innovation, competition, and productivity. Technical report, McKinsey Global Institute, 2011.

[11] M. Muja and D. G. Lowe. Fast approximate nearest neighbors with automatic algorithm configuration. In *International Conference on Computer Vision Theory and Application VISSAPP'09)*, pages 331–340. INSTICC Press, 2009.

[12] S. Ramaswamy and K. Rose. Adaptive cluster distance bounding for high-dimensional indexing. *Knowledge and Data Engineering, IEEE Transactions on*, 23(6):815–830, 2011.

[13] X. Tang, S. Mills, D. Eyers, K.-C. Leung, Z. Huang, and M. Guo. Performance bottlenecks in manycore systems: A case study on large scale feature matching within image collections. In *Proceedings of the 15th IEEE International Conference on High Performance Computing and Communications*, 2013. to appear.

[14] A. Torralba, R. Fergus, and W. T. Freeman. 80 million tiny images: A large data set for nonparametric object and scene recognition. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 30(11):1958–1970, 2008.

[15] R. Weber and K. Böhm. Trading quality for time with nearest-neighbor search. In C. Zaniolo, P. Lockemann, M. Scholl, and T. Grust, editors, *Advances in Database Technology (EDBT)*, volume 1777 of *Lecture Notes in Computer Science*, pages 21–35. Springer Berlin Heidelberg, 2000.

[16] C. Zanchettin, B. L. D. Bezerra, and W. W. Azevedo. A KNN-SVN hybrid model for cursive handwriting recognition. In *Proceedings of the International Joint Conference on Neural Networks*, 2012.