Cost-Efficient Implementation of *k*-NN Algorithm on Multi-Core Processors

Armin Ahmadzadeh, Reza Mirzaei, Hatef Madani, Mohammad Shobeiri, Mahsa Sadeghi,
Mohsen Gavahi, Kianoush Jafari, Mohsen Mahmoudi Aznaveh, Saeid Gorgin*
School of Computer Science, Institute for Research in Fundamental Sciences (IPM), Tehran, Iran.
HPC@ipm.ir

***Abstract*—k-nearest neighbor's algorithm plays a significant role in the processing time of many applications in a variety of fields such as pattern recognition, data mining and machine learning. In this paper, we present an accurate parallel method for implementing k-NN algorithm in multi-core platforms. Based on the problem definition we used Mahalanobis distance and developed mathematic techniques and deployed best programming experiences to accelerate contest reference implementation. Our method makes exhaustive use of CPU and minimizes memory access. This method is the winner of cost-adjust-performance of MEMOCODE contest design 2014 and is 616x faster than the reference implementation of the contest.**

Keywords— k-NN algorithm; Mahalanobis distance; Cost-efficent; Multi-core processors

I. INTRODUCTION

K-nearest neighbors (k-NN) algorithm is one of the most popular algorithms used for pattern recognition, data mining, and machine learning applications [1, 2]. The subject of MEMOCODE 2014 Design Contest is to find k-NN using Mahalanobis distance [3]. The Mahalanobis distance depends on the distribution of points in a feature space. This distance can be defined as a dissimilarity measure between two random vectors $\vec{x}$ and $\vec{y}$ of the same distribution with the covariance matrix S. The calculated distance is based on the following equation where $\vec{x}$ and $\vec{y}$ are D dimension input vectors and S^{-1} is $D \times D$ dimension inverse covariance matrix.

$$dist(\vec{x},\vec{y}) = \sqrt{(\vec{x}-\vec{y})^T S^{-1}(\vec{x}-\vec{y})} \qquad (1)$$

In this contest S^{-1} is a 32×32 matrix and the objective is finding 10 nearest neighbors ($K = 10$). The dimension of all the vectors (input data N and dataset C) in this problem is 32.

Independent data computations in this problem make it a good candidate for multi and many core platforms. Generally, two methods can be used to parallelize this algorithm in a coarse grained processor: 1) one input element of data is given to all threads and dataset is divided among these threads, 2) input of each thread differs and each thread searches all the dataset for their input. The former may introduce several performance barriers. Merging different results of different cores is a sequential task which reduces performance. Moreover, CPU cache is not well utilized in this approach and extra memory bandwidth is required. Therefore, we used the latter and distributed input among cores and computed their distances to each point of dataset in each stage of the algorithm. In addition, we used several optimization techniques such as mathematic optimization, data type optimization, and data path optimization.

The rest of this paper is organized as follows: in Section II the previous works on designed implementation of k-NN algorithm are reviewed. We present our proposed methods in Section III, which is the winner of this year's performance per cost design contest. The experimental results are presented in Section IV. Section V is the conclusion of this paper.

II. RELATED WORKS

There are different approaches for k-NN algorithm, and the simplest one is discussed in [4]. This approach, named linear search, explores all neighbors and computes distances to find nearest points. Obviously, searching a large dataset consumes a huge amount of computing power. Therefore, many approximation algorithms are introduced in order to decrease the computation time and complexity [4]. One of the algorithms for this problem is space partitioning. K-D-B tree [5] and quad tree [6] are data structures for space partitioning. This algorithm partitions the feature space into several regions and each region includes a subset of dataset points. To find K nearest neighbors, it is sufficient to search the regions with maximum possibility of neighbor's existence. Labeling the dataset points into different regions eliminates the necessity of searching the whole dataset, which reduces the computation. Selection of region boundaries only based on space is sensitive to data distribution, thus it may cause unbalanced regions. As a result, to find a specific point, we may have to search the whole dataset. In this case, space partitioning has the overhead of labeling and reduces the performance. Moreover, in case of high dimensional data space, K-D-B tree may not be appropriate [7], [8], [9].

Partitioning schemes based on data distribution has been devised to avoid unbalanced partitioning. K-means clustering [10] is an example of data partitioning that clusters data set into K sets. In this method, K reference points are selected to partition the dataset into K clusters. Data points are within a cluster with nearest reference point. In each cluster, reference points are changed based on the average of the points, iteratively. These iterations are repeated till reference point converge. This approach is appropriate for Euclidean space and can improve performance. R-tree is a data structure for this type of algorithms [4, 11]. However, Ref. [4] argues that for high-dimensional spaces, these kind of data structure degrade performance. Therefore, we used linear search along with best programming experiences to improve the performance of reference implementation.

* S. Gorgin is also affiliated with Iranian Research Organization for Science and Technology (IROST), Tehran, Iran.

III. PROPOSED METHODS

In this section, we present our techniques that reduce the running time of the Mahalanobis algorithm. Profiling reference implementation shows computing Mahalanobis distance is the bottleneck of the algorithm. It is computed by three matrix multiplications (See Equ. (1)). Matrix multiplication needs a huge amount of computation and memory access. These computations include two nested loops and the complexity is $O(n^2)$. Another challenge in this problem is memory access that can be relieved by optimizing memory usage.

In addition, very tight constraint on accuracy does not allow approximation and randomized algorithms. This is due to problem definition that only 2-bit error is allowed in the final results. Therefore, we are limited to certain optimization techniques, which are discussed in the following sections.

A. Mathematic Optimization

We implemented two different schemes to reduce the computation of the algorithm. In one scheme, we mapped the Mahalanobis space to Euclidean space. We need less computation in Euclidean space. This scheme is introduced in this section, part 1. The approach that we actually used is reformulating Mahalanobis distance to decrease the computation. This method helped us to pre-compute a part of formula and avoid re-computation. This approach is presented in this section part 2, 3.

1) Mapping Mahalanobis space to Euclidean space

It is possible to map Mahalanobis space to Euclidean space. This scheme can reduce computation. The mathematical foundation is presented in this part. Comparing square of distances is sufficient for implementing the program, which is also done in naïve approach. This formula is:

$$dist_M^2(\vec{x},\vec{y}) = (\vec{x}-\vec{y})^T S^{-1}(\vec{x}-\vec{y}) \qquad (2)$$

In the above equation S^{-1} is covariance matrix which is defined as follow:

$$S_{ij} = cov(X_i, X_j) = E\left[(X_i-\mu_i)(X_j-\mu_j)\right]$$

Where

$$\mu_i = E(X_i)$$

The S matrix is the expected value of i^{th} entry in the vector X:

$$S = \begin{bmatrix} E[(X_1-\mu_1)(X_1-\mu_1)] & \dots & E[(X_1-\mu_1)(X_n-\mu_n)] \\ E[(X_2-\mu_2)(X_1-\mu_1)] & \dots & E[(X_2-\mu_2)(X_n-\mu_n)] \\ \vdots & \ddots & \vdots \\ E[(X_n-\mu_n)(X_1-\mu_1)] & \dots & E[(X_n-\mu_n)(X_n-\mu_n)] \end{bmatrix}$$

The S matrix is a symmetric matrix. It is also a positive semi-definite matrix which can be mathematically explained as follows:

M is a positive-semidefinite if $x^*Mx \geq 0$ for all x in $\mathbb{C}^n$(or, all x in $\mathbb{R}^n$ for the real matrix).

All positive semi-definite matrices can be decomposed using eigenvalues:

$$M = VPV^{-1}$$

Where P is a diagonal matrix in which the diagonal entries are eigenvalues of M. V is one of corresponding eigenvectors of M. Evidently, the matrix P always has an inverse:

$$P = \begin{bmatrix} d_1 & 0 & \dots & 0 \\ 0 & d_2 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & d_n \end{bmatrix}, P^{-1} = \begin{bmatrix} \frac{1}{d_1} & 0 & \dots & 0 \\ 0 & \frac{1}{d_2} & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & \frac{1}{d_n} \end{bmatrix}$$

Semi-definite matrix' eigenvectors have a unique characteristic that are equal to their transpose:

$$V^T = V^{-1}$$

Now we can rewrite the Mahalanobis distance equation as follows:

$$\begin{aligned} dist_M^2(\vec{x},\vec{y}) &= (\vec{x}-\vec{y})^T(VPV^T)^{-1}(\vec{x}-\vec{y}) \\ &= (\vec{x}-\vec{y})^T VP^{-1}V^T(\vec{x}-\vec{y}) \\ &= (\vec{x}-\vec{y})^T VP^{-1}V^T(\vec{x}-\vec{y}) \\ &= (\vec{x}-\vec{y})^T VP^{-1/2}P^{-1/2}V^T(\vec{x}-\vec{y}) \end{aligned} \qquad (3)$$

Comparing (2) and (3), we can discover that matrix S^{-1} have a square root matrix, $S^{-1/2}$, hence

$$\begin{aligned} dist_M^2(\vec{x},\vec{y}) &= (\vec{x}-\vec{y})^T S^{-1/2}S^{-1/2}(\vec{x}-\vec{y}) \\ &\xrightarrow{S^{-1/2}=(S^{-1/2})^T} (\vec{x}-\vec{y})^T\left(S^{-1/2}\right)^T S^{-1/2}(\vec{x}-\vec{y}) \\ &\xrightarrow{(AB)^T=B^TA^T} \left[S^{-1/2}(\vec{x}-\vec{y})\right]^T\left[S^{-1/2}(\vec{x}-\vec{y})\right] \end{aligned} \qquad (4)$$

The matrix $\left[S^{-1/2}(\vec{x}-\vec{y})\right]$ is actually a vector and (4) is the exact format of computing distances in Euclidean space. Computing dot product of two vectors needs less computing power than matrix computation and the complexity of bottleneck of the algorithm is reduced to $O(n)$.

We have implemented the algorithm using the above computation. Due to imprecise floating point operation, unfortunately this approach is not fruitful; both the performance decreases and the final result is not accurate.

2) Mahalanobis distance reformulating

We could find techniques that manipulate formulas in order to improve performance. Let $\vec{x}$ be the input and $\vec{y}$ the dataset. We can reformulate the Mahalanobis distance formula as:

$$\begin{aligned} dist_M^2(\vec{x},\vec{y}) &= (\vec{x}-\vec{y})^T S^{-1}(\vec{x}-\vec{y}) \\ &\xrightarrow{(A\pm B)^T=A^T\pm B^T} (\vec{x}^T-\vec{y}^T)S^{-1}(\vec{x}-\vec{y}) \\ &= (\vec{x}^TS^{-1}-\vec{y}^TS^{-1})(\vec{x}-\vec{y}) \\ &= \vec{x}^TS^{-1}\vec{x}-\vec{y}^TS^{-1}\vec{x}-\vec{x}^TS^{-1}\vec{y}+\vec{y}^TS^{-1}\vec{y} \end{aligned}$$

$\vec{x}^TS^{-1}\vec{x}$ is a scalar value. $\vec{x}^T$ is $1\times D$ matrix, S^{-1} is a $D\times D$ matrix, $\vec{x}$ is $D\times 1$ matrix. $\vec{x}^TS^{-1}\vec{y}$ and $\vec{y}^TS^{-1}\vec{x}$ are scalars and they are equal, $\vec{x}^TS^{-1}\vec{y} = \vec{y}^TS^{-1}\vec{x}$. The equality can be easily seen by expanding matrices. So, the Mahalanobis distance can be rewritten as:

$$dist_M^2(\vec{x},\vec{y}) = \vec{x}^TS^{-1}\vec{x} - 2\vec{x}^TS^{-1}\vec{y} + \vec{y}^TS^{-1}\vec{y}$$

$\vec{y}^TS^{-1}\vec{y}$ is also a scalar. This scalar can be pre-computed. This method reduces D dimensional computation to just a scalar pre-computation and decreases memory access about 32 times. $\vec{x}^TS^{-1}\vec{x}$ is executed N times and is not a large portion of the computations. The $S^{-1}\vec{y}$ part of $\vec{x}^TS^{-1}\vec{y}$ can also be pre-computed and $\vec{x}^TS^{-1}\vec{y}$ is executed $|C|$ times for each input. This part is the bottleneck of the algorithm and consumes most

of the computing power. As a short summery of this part, we shifted a great time of computation to pre-compute time, reduced one multiplication and decreased $N \times |C|$ times computation to just N times ($|C| \gg N$).

3) Using mathematic optimization

S^{-1} is a covariance matrix which is symmetric and needs less storage. This feature can help us reduce memory access and the computation also can be reduced, however the complexity of the algorithm does not change. Consider the naïve implementation of matrix multiplication:

$$(xS^{-1})_i = \sum_{j=1}^{n} x_j S_{ij}^{-1}, i = 1,\dots,n \quad (5)$$

The complexity of above equation is $O(n^2)$. While the entries of a symmetric matrix are equal with respect to the main diagonal, the (5) can be rewritten with two partial sums:

$$(P_1)_i = 2 \times \sum_{j=i+1}^{n} x_j S_{ij}^{-1}, i = 1,\dots,n \quad (6)$$
$$(P_2)_i = \sum_{j=1}^{n} x_j S_{jj}^{-1}, i = 1,\dots,n \quad (7)$$
$$(xS^{-1})_i = (P_1)_i + (P_2)_i \quad (8)$$

Equation (8) equals to the result of (5) which is sum of partial additions of (6) and (7). Equation (7) is applied to diagonal entries and (6) can be applied to either upper triangular or lower triangular, while S^{-1} is symmetric. Time complexity of these equations is roughly $O(n^2/2)$, thus the computation is virtually reduced by half.

B. Data Type Optimization

In the reference implementation, input data and dataset variables are 64 bits wide, however based on problem definition input variables need just 12 bits. This memory assignment is not appropriate and consumes extra memory bandwidth, and reduces performance.

There are three main stages in our algorithm, in which data types need different sizes. Using proper data size in each stage let the compiler use proper SIMD feature of the platform. The first stage is a simple subtraction that does not change the size of the partial result. While general purpose processors have certain data types (such as short int, int or long int), we store variables of first stage in 16 bits. In second stage, the result of previous stage is multiplied by covariance matrix (S^{-1}). Result of this multiplication makes a vector that contains 32 elements. Each element of this vector is produced by multiplication of two 12 bits value, a 24 bits value. This operation is repeated for each element, and then the results of these operations are accumulated. While each vector contains 32 elements and each element is 24 bits, addition of all elements need $24 + \log(32)$ bits (29 bits). We have to store the result of this stage in a 32 bit variable. In the last stage, the dot product of prior stage vector and the difference vector is computed. There is a multiplication of 29 bits by 12 bits vector which is a 41 bits vector. There are 32 additions of these 41 bits vector which results in a 46 bits variable. Due to restraints of platform architecture, we store the last value in a long integer data type, which consumes 64 bits.

The proper usage of data types that mentioned in last paragraph reduces overall memory usage from 2.8 GB to 800 MB. Using this method allows us to implement the algorithm in platforms that have less RAM, like GPUs and FPGAs. Reducing memory usage by about one third leads not only to a better memory access, but also to a better consumption of memory bandwidth. Moreover, cache miss is reduced.

C. Data Path Optimization.

We have used a passing technique in order to avoid some parts of computation. The $2\vec{x}^T S^{-1}\vec{y}$ is the part that we have to calculate $|C|$ times for each input. To compute this part we need a loop, with D times iteration, for each element of dataset. We observed that many elements of dataset are far from the intended input and this can be understood after a few iterations in the loop. Therefore, we compare the distance with the furthest among the K nearest neighbors computed so far and bypass the remaining part of the loop, if the current data element cannot be enlisted in nearest neighbors. The pseudo code of this part is presented here:

Algorithm 1: data path optimization

```
1:  Dist ← x^T S^-1 x + y^T S^-1 y ;
2:  dist_temp ← 0; f ← 0; max_try ← 1;
3:  FOR i =1 TO D STEP 4 DO
4:      FOR k = 1 TO 4 DO
5:          dist_temp ← dist_temp + (x^T_(i+k)) × S^-1 × y_(i+k)) ;
6:      END DO
7:      IF ((Dist − 2 × dist_temp) > dist_lowest[k − 1]) THEN
8:          f ← f + 1;
9:          IF ( f > max_try)
10:             RETURN;
11:     ELSE
12:         f ← 0;
13:     END IF
14: END DO
15: dist_new = Dist − 2 × dist_temp;
16: Sort (dist_new); // sort new distance between K nearest
17: RETURN;
```

We devised an approximation method to be pretty confident that the current point will not be in nearest neighbors set. Note that while there are both positive and negative numbers in the addition of the algorithm, there may always be scenarios that the result of addition changes in the last iteration. However, we observed that prediction of outcome is admissible.

The method that we use is obtained from the branch prediction technique commonly used in modern processors. After each 4 iterations of the loop, we compare the results with nearest neighbors list (line 7 in algorithm 1). If the result is greater than the greatest element in the set of K nearest neighbors, then we change the state of our current situation. The overall diagram of this action can be seen in the Fig. 1.

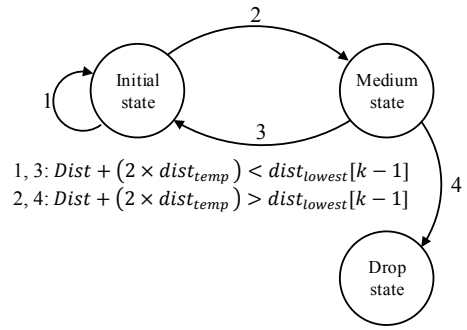

Fig. 1 FSM of prediction method

We bypass the rest of the loop if the program enters in the drop state (line 10 in algorithm 1). We observed many of dataset points need just 8 iterations and the remaining 24 iteration can be bypassed.

IV. EXPERIMENTAL RESULT

MEMOCODE hardware/software co-design contest is in two classes: absolute-performance and cost-adjusted-performance. We implemented our algorithm on multi-core systems. Here is a list of our multi-core platforms:

- Intel Core i5, 2410M with two cores @ 2.30GHz.
- Intel Core i7, 960 with four cores @ 2.67GHz.
- Intel Xeon x5650 with six core @ 2.67GHz.
- Intel Xeon E5-2650 with eight core @ 2.00GHZ.
- Intel Xeon Phi 5110P with sixty core @ 1.053 GHz

We took advantage of multi-core platforms via OpenMP programming. To gain better efficiency, we used Intel compiler and OpenMP4.0. For the performance evaluation, we implemented our approach in various multi core platforms. Even though error in two LSBs was acceptable, our program was executed without any error in the reported results.

We tested our implementation on five multi-core platforms. The platforms, prices, and execution time on each platform for the large dataset (input data $N = 1{,}000$ and dataset $C = 10{,}000{,}000$) are shown in Table 1.

TABLE I. PRICES, AND EXECUTION TIME FOR EACH PLATFORM

Design	Intel Platform	Time (S)	Cost (US$)	Performance × Cost	Speed Up
Naive	Xeon 2650	9240	1016	9387748	1 X
Proposed Solution	corei5	166	35.99	5975	56 X
	corei7	71	100	7100	130 X
	Xeon 5650	35	160	5600	264 X
	Xeon 2650	**15**	**1016**	**15240**	**616 X**
	Xeon Phi 5110P	54	729	39366	171 X

Based on the presented results in the above table, the best platform in terms of performance per cost is Intel Xeon 5650; the best speed up of these platforms is 616X. Summary of all the result are presented in Fig. 2.

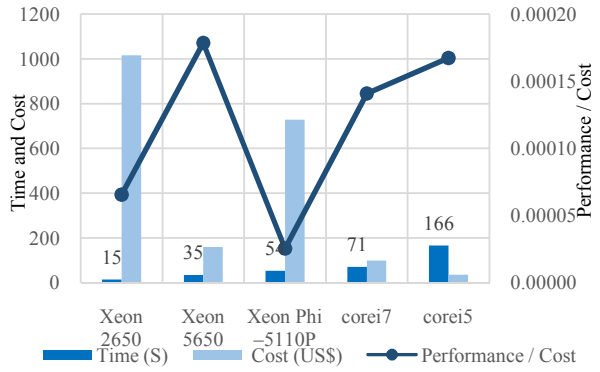

Fig. 2 Time, price and performance/cost amount for various multi-core processors

V. CONCLUSION

In this paper, we presented a CPU implementation of k-nearest neighbor algorithm based on the Mahalanobis distance. Our CPU implementation is 616 times faster than the naïve approach of the contest. Our implementation deploys memory reduction and decreases bandwidth requirements of k-NN with Mahalanobis distance. We also manipulate some parts of algorithm to bypass inessential computations and accelerate overall execution process. Although two bits deviation error is allowed in the definition of the contest problem, our approach is 100% accurate. Our implementation is the winner of cost-adjusted-performance of MEMOCODE 2014 Design Contest.

ACKNOWLEDGMENT

We are grateful to Prof. Hamid Sarbazi Azad, Head of the school of computer science, for his support and useful guidance. We also would like to acknowledge all members of computer science school at the Institute for Research in Fundamental Sciences (IPM) for their helpful discussions and comments.

REFERENCES

[1] R. Chatpatanasiri, T. Korsrilabutr, P. Tangchanachaianan, and B. Kijsirikul, "A new kernelization framework for Mahalanobis distance learning algorithms," *Neurocomputing,* vol. 73, pp. 1570-1579, 2010.

[2] K. Q. Weinberger, J. Blitzer, and L. K. Saul, "Distance metric learning for large margin nearest neighbor classification," in *Advances in neural information processing systems*, 2005, pp. 1473-1480.

[3] *MEMOCODE 2014 Design Contest*. Available: http://memocode.irisa.fr/2014/

[4] R. Weber, H.-J. Schek, and S. Blott, "A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces," in *VLDB*, 1998, pp. 194-205.

[5] J. T. Robinson, "The KDB-tree: a search structure for large multidimensional dynamic indexes," in *Proceedings of the 1981 ACM SIGMOD international conference on Management of data*, 1981, pp. 10-18.

[6] R. A. Finkel and J. L. Bentley, "Quad trees a data structure for retrieval on composite keys," *Acta informatica,* vol. 4, pp. 1-9, 1974.

[7] H. Jagadish, B. C. Ooi, K.-L. Tan, C. Yu, and R. Zhang, "iDistance: An adaptive B+-tree based indexing method for nearest neighbor search," *ACM Transactions on Database Systems (TODS),* vol. 30, pp. 364-397, 2005.

[8] C. Yu, B. C. Ooi, K.-L. Tan, and H. Jagadish, "Indexing the distance: An efficient method to knn processing," in *VLDB*, 2001, pp. 421-430.

[9] J. E. Goodman and J. O'Rourke, *Handbook of discrete and computational geometry*: CRC press, 2010.

[10] J. MacQueen, "Some methods for classification and analysis of multivariate observations," in *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability, Volume 1: Statistics*, Berkeley, Calif., 1967, pp. 281-297.

[11] A. Guttman, *R-trees: a dynamic index structure for spatial searching* vol. 14: ACM, 1984.