Implementation of a Parallel K-Nearest Neighbor Algorithm Using MPI

Ahmed S. J. Abu Hammad

***Abstract*—**The K-Nearest Neighbor (K-NN) algorithm is one of the most commonly used algorithms for Classification. The traditional K-NN algorithm is; however, inefficient while working with large data sets because the computation cost is quite high as we need to compute the distance of each query instance to all training samples and sorting the distances to determine the nearest neighbors. This paper presents a parallel implementation of the K-NN algorithm using Message Passing Interface (MPI) by distributing the computations of the distance value operation to different processors. Also, we focus on improving the performance of sequential K-NN algorithm with proposed Parallel K-NN (PK-NN) algorithm. We applied our experiments on the graduate student's data set collected from the University College of Science and Technology (UCST) – Khan Younis. We compute the execution time, speedup, efficiency, parallel overhead, and parallel cost and discuss our results for serial and parallel algorithms by comparing them. Finally, we find parallel version reduce the time and more effective while applied in large data set than the sequential one.

***Keywords*—**Classification, k-NN algorithm, Parallel classifier, Parallel and distributed computing, Multicomputer cluster.

I. Introduction

Data mining is to discover interesting, meaningful and understandable patterns hidden in massive data sets. Traditional knowledge discovery systems have been found lacking in their ability to handle current data sets, due to characteristics such as their large sizes and high dimensional. Consequently, new techniques that can automatically transform these large data sets into useful information are in strong demand [2,3].

Classification is one of the important data mining techniques whereby a model is trained on a data set with class labels and then used to predict the class label of unknown objects. K-NN is one of the most widely used techniques in classification applications, where the result of the new instance query is classified based on the majority of K-NN category. The purpose of this algorithm is to classify a new object based on attributes and training samples [2].

The K-NN algorithm is very easy to implement and can produce good results, but the computation cost is quite high because we need to compute the distance of each query instance to all training samples and sorting the distances to determine the nearest neighbors. Therefore, parallel computing is an essential component of the solution to speed up K-NN [3].

In this paper, we present a parallel implementation of the K - NN algorithm using MPI and c programming language to accelerate the distance computation and sorting, and we conduct an experiment to study the performance of parallelizing K-NN and compare it with the serial K-NN version as a baseline.

The rest of this paper is organized as follows: Section 2 presents related works. Section 3 describes the PK-NN algorithm model. Section 4 presents the proposed method. Finally, a P-KNN implementation and experimental results, and conclusions are presented in Section 5 and Section 6 respectively.

Ahmed Abu Hammad, University College of Science and Technology, Khan Younis, Palestine (e-mail: asj_hammad@hotmail.com).

II. RELATED WORKS

Serial K-NN Algorithm: The k-nearest-neighbor method [2] was first described in the early 1950s. The method is labour intensive when given large training sets, and did not gain popularity until the 1960s when increased computing power became available. It has since been widely used in the area of pattern recognition and text mining.

Nearest-neighbor classifiers are based on learning by analogy, that is, by comparing a given test tuple with training tuples that are similar to it. The training tuples are described by *n* attributes. Each tuple represents a point in a *n*-dimensional space. In this way, all the training tuples are stored in a *n*-dimensional pattern space. When given an unknown tuple, a K-NN classifier searches the pattern space for the *k* training tuples that are closest to the unknown tuple. These *k* training tuples are the *k* nearest neighbors of the unknown tuple. "Closeness" is defined in terms of a distance metric, such as Euclidean distance. The Euclidean distance between two points or tuples, $X=(x_1,x_2,\ldots,x_n)$ and $Y=(y_1,y_2,\ldots,y_n)$ is:

$$d(X,Y) = \sqrt{\sum_{i=1}^{n}(x_i + y_i)^2} \qquad (2.1)$$

The standard K-NN algorithm can be described as follows [2]:

1. Determine a parameter *K* = number of the nearest neighbors.
2. Calculate the distance between the query-instance and all the training samples.
3. Sort the distance and determine the nearest neighbors based on the *K-th* minimum distance
4. Gather the category of the nearest neighbors
5. Use a simple majority of the category of the nearest neighbors at the prediction value of the query instance.

Few attempts have been made to parallelize K-NN to increase its speed.

Lianga et al. [3] implemented a parallel learning algorithm. The parallel algorithm is based on the k-NN algorithm. They evaluated the parallel implementation on Compute Unified Device Architecture (CUDA) enabled Graphics Processing Unit (GPU). The advantage of this method is the highly parallelizable architecture of the GPU. Recent development in GPUs has enabled inexpensive high-performance computing for general-purpose applications. Due to GPU's tremendous computing capability, it has emerged as the co-processor of the Central Processing Unit (CPU) to achieve a high overall throughput. CUDA programming model provides the programmers adequate C language like APIs to better exploit the parallel power of the GPU and manipulate it. At the hardware level, CUDA-enabled GPU is a set of Single Instruction Stream, Multiple Data Stream (SIMD) processors with 8 stream processors. They used synthetic data generated by MATLAB for the purpose of evaluation where the number of data objects is 262144 records. Their experiment showed good scalability on data objects. CUK-NN presented up to 15.2 speedup. The result shows that CUK-NN is suitable for large scale dataset. However, since SIMD processors are specially designed, they tend to be expensive and have long design cycles and the scalability of the processors is limited.

Tarhini [3] implement a parallel K-NN method on the CPU rather than a GPU where the degree of parallelism is indicated by the number of available cores. The proposed algorithm is not expected to outperform state-of-the art GPU implementations, but rather, to provide an equivalent performance on the CPU. Hence, the benefit becomes the ability of load sharing between CPU and GPU without degradation or loss of speed upon switching between any of the two processor architectures. The experiment was implemented on an Intel 8-core machine in which the cores are integrated onto a single integrated circuit die (known as a chip multiprocessor). Iris database was used to train the system. The data set contains 50,000 records. The parallel implementation greatly increased the speed of the KNN algorithm by reducing its

time complexity from *O(D)*, where *D* is the number of records, to *O(D/p)* where *p* is the number of processors.

However, our platform comprises a set of processors and their own exclusive memory (multi-computer workstation cluster), this platform is programmed using send and receive primitives, Message Passing Interface (MPI) provide such primitives.

III. P-KNN Algorithm Model

An algorithm model is a way of structuring a parallel algorithm by selecting a decomposition and mapping technique.

3.1 Decomposition Technique

The first step in developing a parallel algorithm is to decompose the problem into tasks that can be executed concurrently by identifying the data on which computations are performed, then partition this data across various tasks [1].

The task performs the computations with its part of the data. In our algorithm, the input data partitioning is the natural decomposition technique because the output (the computed distances) is not clearly known a-priori. It divides the data set equally according to the number of worker processes by sending a one data partition for each of them [1].

3.2 Mapping Technique

Once a problem has been decomposed into concurrent tasks, these must be mapped to processes (that can be executed on a parallel platform) [1].

In our algorithm, we use the static mapping technique that distributes the tasks among processes prior to the execution of the algorithm.

The scheme for this static mapping is mapped based on data partitioning because our data represented in a two-dimensional array. So, the most suitable scheme used for distributing the two-dimensional array among processes is the row-wise *1-D* block array distribution that distributes the array and assign uniform contiguous portions of the array to different processes [1].

According to the previous selected decomposition and mapping techniques, the suitable parallel algorithm model is the master-slave model in which the master process generates the work and assigns it to worker processes.

3.3 Communication Operation

In most parallel algorithms, processes need to exchange data with other processes; in our algorithm the master process divide the data set according to the number of workers and sending a one data partition for each of them with the *j* and query-instance values. Also, the worker processes send the *j-th* ordered list to the master which include the *j-th* distances and classes.

The suitable communication operation for our algorithm is the scatter operation (one-to-all personalized communication), in which the master process sends a unique part of the divided data to each of the worker processes [1].

The dual of one-to-all personalized communication or the scatter operation is the gather operation, or concatenation, in which the master process collects a unique *j-th* ordered list from each other worker processes [1].

IV. The Proposed Method

Since the computation of the distance between the input sample and any single training sample is independent of the distance computation to any other sample, that allows for partitioning the computation work with the least synchronization effort. In fact, no intercommunication or message passing is required at all during the time each processor is computing the distance between samples in its local storage and the input sample. When all processors terminate the distance computation procedure, the final step is to select a master processor to collect the results from all processors, sort the distances in ascending order, and then use the first *j* measures to determine the class of the input sample.

The proposed algorithm is described in the following steps:

1. Reading the training data set, *j* value and the test object attribute values. Then, determining one process as a master process and the remaining processes as workers.
2. The master process divides the data set equally according to the number of workers by sending a one data partition for each of them with the *j* and query-instance values.
3. Each worker process receives its data partition and the other parameters then:
    a. Calculate the distance between the query-instance and all the training samples.
    b. Sort the distance and determine nearest neighbors based on the *j*-th minimum distance locally.
    c. Send the *j*-th ordered list to the master which include the *j*-th distances and classes.
4. The master process receives from each worker the *j*-th ordered list and combining them in a *j*-th master list.
5. The master now:
    a. Sort the *j*-th master list elements in ascending order.
    b. Select the *j*-th top element.
    c. Compute the majority of classes in the top *j*-th element.
    d. Define the query-instance according to the major class.

Figure 1 exhibits the proposed solution:

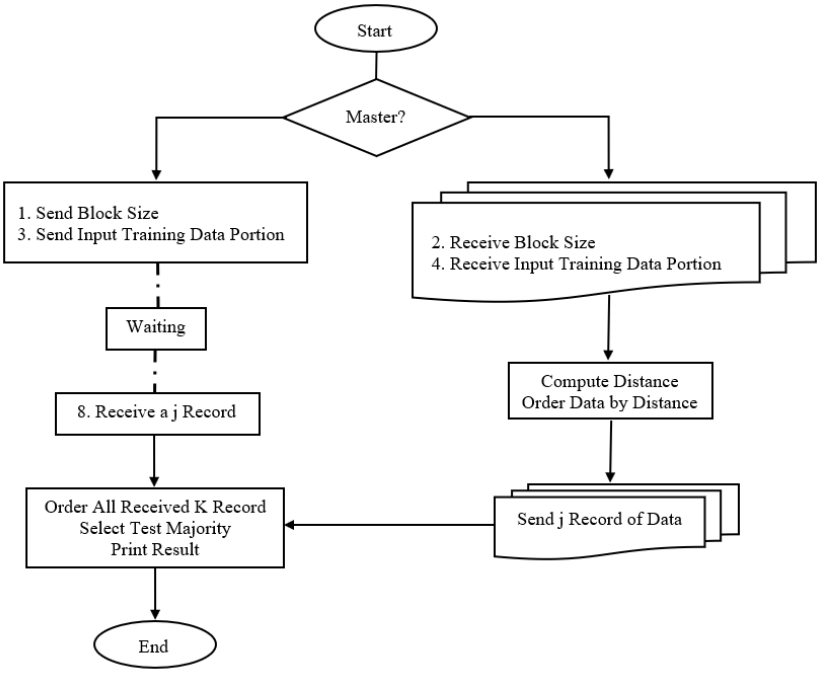

Fig. 1 The Flow chart of the proposed solution.

V. P-KNN Implementation and Experimental Results

We implemented the sequential and parallel versions of the K-NN algorithm and apply our experiments on graduate student's dataset collected from the University College of Science and Technology in Khan Younis. Also, we compare the results between the sequential and parallel versions of the K-NN algorithm, then determine the time in the two cases, and we calculate the speed up, efficiency, parallel overhead, and parallel cost.

For our experiment, the tests are done in a cluster which has consisted of 8 workstations, all workstations have the same specification; Intel(R) Core (TM)2 Quad CPU Q8300 @ 2.50 GHz, 4.00 GB RAM, 320 GB hard disk drive and Windows 7 operating system installed, using the parallel message passing software MPICH2

5.1 Data Collection

The dataset used in this paper contains graduate student's information collected from the University College of Science and Technology – Khan Younis for a period of fifteen years in the period from 1993 to 2011. The graduate student's dataset consists of 17000 records and 18 attributes. Each record belongs to 1 of 5 categories (Excellent – Very Good – Good – Acceptable – Fail). Table I presents the attributes and their description that exist in the data set as taken from the source database.

TABLE I
THE GRADUATE STUDENT'S DATASET DESCRIPTION

Attribute	Description	Data Type	Selected
ID	An identifier for the record	Number	
Name	Student's named	String	
DOB	Date of birth	String	
Gender	Student's gender	String	√
Nationality	Student's nationality	String	
City	Student's address details	String	√
GS Source	Source general secondary	String	
GS Year	Year general secondary	Number	
GS Avg	Average general secondary	Number	√
GS Sec	Section general secondary	Number	√
YI Join	Year institution join	Number	
YI Term	Year institution term	Number	
Std Level	Student's level	Number	
College	Student's college	String	
Specialization	Student's specialization	String	√
HC Num	Hours Completed number	Number	
GPA	Student's a cumulative grade point average	Number	
Grade	Student's performance	String	√

As part of the preparation and pre-processing of the data set, irrelevant and weakly relevant attributes should be removed. The attributes marked as selected as seen in Table 1 are processed via the rapid miner software to apply the data mining methods on them.

5.2 Experimental Results and Discussion

We execute the K-NN (sequential/parallel) program on the training data set with varying number of processes and problem size to evaluate the performance. We run our experiment on 3 to 8 processors, and a problem size with 5000, 13000, and 17000 records, then we compared them with the sequential version. For sequential method, we compute the time using a clock () method from time.h library. For the parallel method we use MPI_Wtime() from mpi.h library that returns an elapsed time on the calling processor in seconds. The execution time in seconds is recorded in Table II.

TABLE II
THE EXECUTION TIME OF THE SEQUENTIAL AND PARALLEL CLASSIFIERS.

Problem Size / PNum		5000	13000	17000
Sequential K-NN		0.07800	0.37500	0.56200
PKNN with No. of processes	*3-process*	0.04610	0.16076	0.22337
	4-process	0.03740	0.15242	0.15924
	5-process	0.03285	0.14024	0.13518
	6-process	0.03093	0.12872	0.11934
	7-process	0.03008	0.11769	0.10394
	8-process	0.02814	0.10967	0.09854

As we note from Table II, the sequential version takes more time than the parallel version. In the parallel version; the execution time decreases when the number of processes increases. However, the parallel implementation achieves a good execution time compared to the sequential one. Figure 2 illustrates the execution time. The time curve decreases from 3 processors until use 8 processors

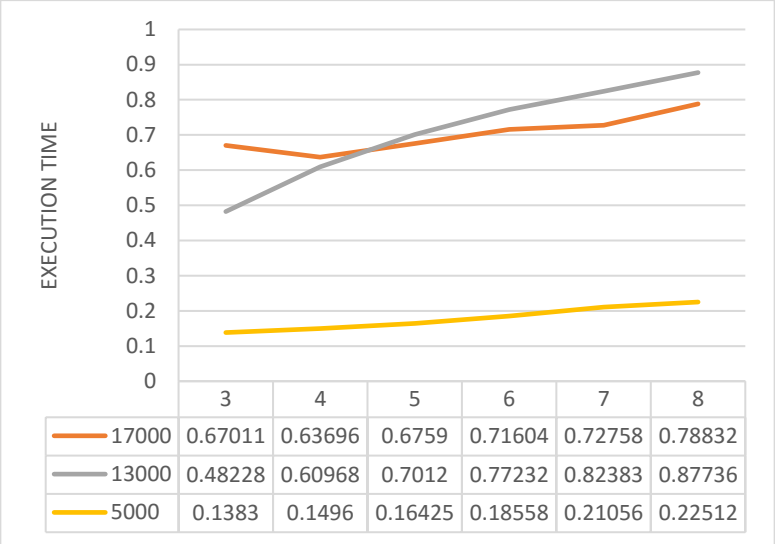

Fig. 2 The Curves of Execution Time for the Classifiers.

Also, we compute Speedup (S) which gained from this parallelization by using equation 1. Speedup values are recorded in Table III.

$$S = T_S / T_P \qquad (5.1),$$

where T_S is serial time and T_P is parallel time.

TABLE III
THE RELATIVE SPEEDUP OF THE PROPOSED PARALLEL CLASSIFIER.

Problem Size / PNum	5000	13000	17000
3-process	1.69197	2.33267	2.51600
4-process	2.08556	2.46031	3.52926
5-process	2.37443	2.67399	4.15742
6-process	2.52182	2.91330	4.70923
7-process	2.59309	3.18634	5.40697
8-process	2.77186	3.41935	5.70327

As we note from Table III, when increasing the number of processors, the Speedup values tend to be saturated, also we note that the Speedup values less than the number of processing elements. Figure 3 illustrates the speed up curve.

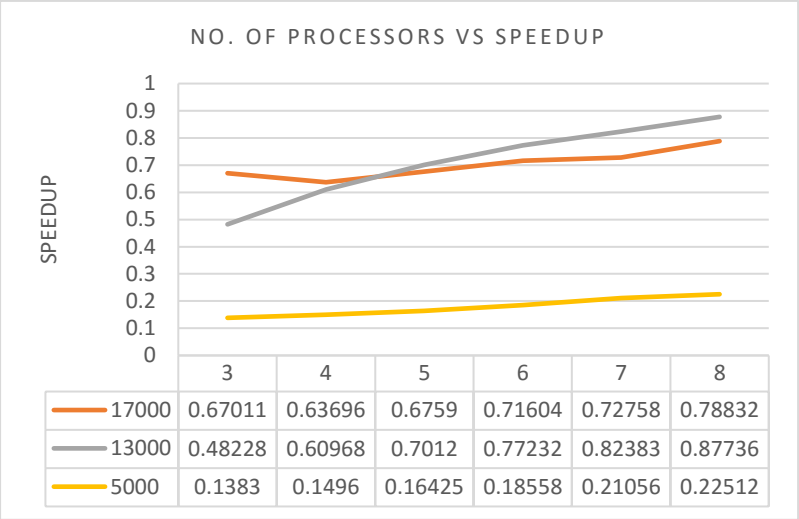

Fig. 3 The Relative Speedup Curves of the Proposed Parallel Classifier.

From the Speedup computation, we can compute the efficiency (E) using equation 2. Efficiency values are recorded in Table IV.

$$E = S/P \qquad (5.2),$$

where S is the Speedup and P is the number of processing elements.

TABLE IV
THE EFFICIENCY OF THE PROPOSED PARALLEL CLASSIFIER.

PNum \ Problem Size	5000	13000	17000
3-process	0.56399	0.77756	0.83867
4-process	0.52139	0.61508	0.88232
5-process	0.47489	0.53480	0.83148
6-process	0.42030	0.48555	0.78487
7-process	0.37044	0.45519	0.77242
8-process	0.34648	0.42742	0.71291

As we note from Table IV, the values of the efficiency are between 0 and 1 and we note that the efficiency decreases as the number of processing elements is increased for a problem size to 5000, and 13000 records and this is common to all parallel programs, but the efficiency where the problem size is 17000 increased when we use 4 processing elements, and after that is decreased, so we can induce that the suitable number of processing elements of our problem is 4. Also, we note that the efficiency increases if the problem size is increased while keeping the number of processing elements constant. Figure 4 illustrates the efficiency curve.

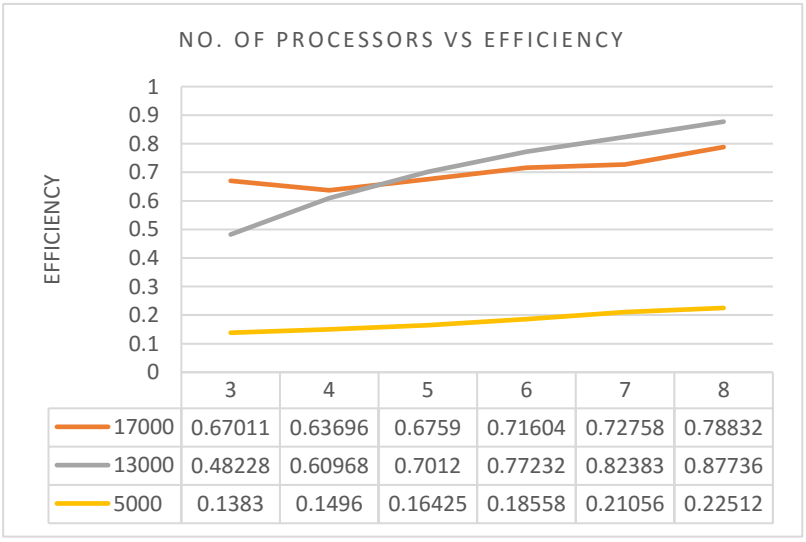

Fig. 4 The Efficiency Curves of the Proposed Parallel Classifier.

Also, we compute parallel Overhead (TO) which is the total time to spend by all processors combined in non-useful work by using equation 3. Overhead values are recorded in Table V.

$$\text{parallel Overhead} = PT_P - T_S \quad (5.3),$$

where T_S is serial time, T_P is parallel time and P is the number of processing elements

TABLE VI
THE SERIAL AND PARALLEL COST.

	Problem Size / PNum	5000	13000	17000
Serial cost		0.07800	0.37500	0.56200
Parallel cost	*3-process*	0.13830	0.48228	0.67011
	4-process	0.14960	0.60968	0.63696
	5-process	0.16425	0.70120	0.67590
	6-process	0.18558	0.77232	0.71604
	7-process	0.21056	0.82383	0.72758
	8-process	0.22512	0.87736	0.78832

As we note from Table V, when increasing the number of processors, the values of the parallel overhead increases for a problem size with 5000, and 13000 records, but the parallel overhead where the problem size is 17000 decreased when we use 4 processing elements, and after that is increased. Figure 5 illustrates the parallel overhead curve.

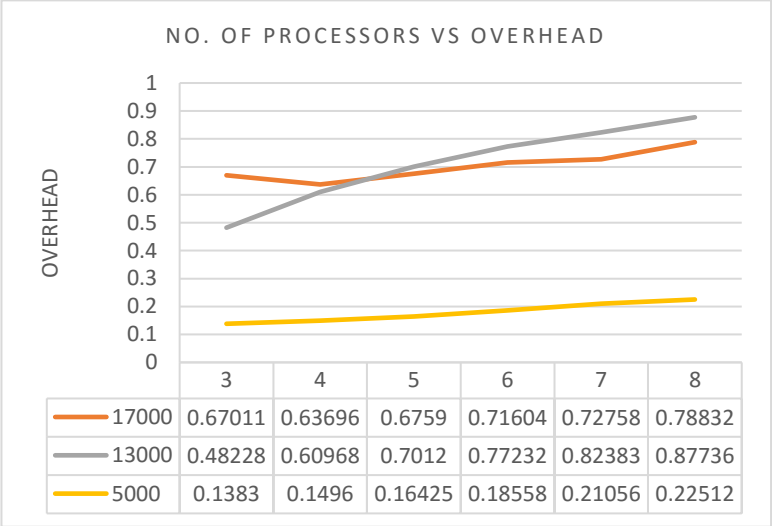

Fig. 5 The Parallel Overhead of the Proposed Parallel Classifier.

Finally, we compute the parallel cost which is the total time collectively spent by all the processing elements using equation 4. Serial & parallel cost values are recorded in Table VI.

$$\text{parallel Cost} = PT_P \quad (5.4),$$

where T_P is parallel time and P is the number of processing elements.

TABLE V
THE PARALLEL OVERHEAD OF THE PROPOSED PARALLEL CLASSIFIER.

Problem Size / PNum	5000	13000	17000
3-process	0.06030	0.10728	0.10811
4-process	0.07160	0.23465	0.07496
5-process	0.08625	0.32620	0.11390
6-process	0.10758	0.39732	0.15404
7-process	0.13256	0.44883	0.16558
8-process	0.14712	0.50236	0.22632

As we note from Table VI, when increasing the number of processors, the values of the

parallel cost increases for a problem size to 5000, and 13000 records, but the parallel cost where the problem size is 17000 decreased when we use 4 processing elements, and after that is increased. Figure 6 illustrates the parallel cost curve.

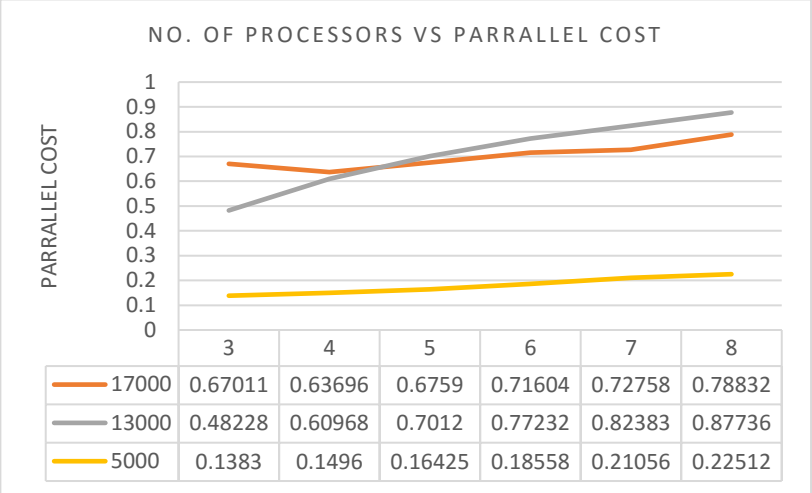

Fig. 6 The serial and parallel cost curve.

To ensure that the classifier works well with the tested records, we also examined the quality of the classification. We split the dataset into two parts (90% of the dataset for training and the remaining 10% to test).

For the purpose of evaluating the classification results, we use confusion matrices that are the primary source of performance measurement for the classification problem. Each column of the confusion matrix represents the instances in an actual class, while each row represents the instances in a predicted class. The K-NN classifier obtains 54.37% accuracy as the best results the number of neighbors is 5.

VI. Conclusion

In this paper, we parallelize the K-NN classification algorithm using message passing routines. We applied our experiments on (5000, 13000, and 17000) × 6 data matrices, by use 3 to 8 processors, and compared them with the sequential version by calculating the execution time for them, we observe in parallel version that the execution time decreases when the number of processes increases. However, the parallel implementation achieves a good execution time compared to the sequential one. We note that when compute Speedup from serial time and parallel time, its values tends to be saturated, also we note that the values less than the number of processing elements. We note that when compute the efficiency, its values are between 0 and 1 and we note that the efficiency decreases as the number of processing elements is increased for a problem size to 5000, and 13000 records and this is common to all parallel programs, but the efficiency where the problem size is 17000 increased when we use 4 processing elements, and after that is decreased, so we can induce that the suitable number of processing elements of our problem is 4. Also, we note that the efficiency increases if the problem size is increased while keeping the number of processing elements constant. Also, we note when increased the number of processors, the values of the parallel overhead increased for a problem size with 5000, and 13000 records, but the parallel overhead where the problem size is 17000 decreased when we use 4 processing elements, and after that is increased.

Finally, we note when increasing the number of processors, the values of the parallel cost increases for a problem size with 5000, and 13000 records, but the parallel cost where the problem size is 17000 decreased when we use 4 processing elements, and after that is increased,

also, we note that from [3] and [4] MPI is not so good as CUDA or using a computer with a multicore microprocessor.

References

[1] Grama A., Gupta A., Karypis G. and Kumar V., "*Introduction to Parallel Computing*", 2nd edition, Addison Wesley, 2003.

[2] Han, J. and Kamber, M. "*Data Mining: Concepts and Techniques*", 2nd edition. The Morgan Kaufmann Series in Data Management Systems, Jim Gray, Series Editor, 2006.

[3] Lianga, S., Liua, Y., Wangb, C., and Jiana, L.CUKNN: "*A parallel Implementation of k-Nearest Neighbor on Cuda-Enabled GPU*", The 2009 IEEE Youth Conference on Information, Computing and Telecommunication (ICT2009) – Conference Proceedings, pp. 415-418, 2009.

[4] Tarhini, A. "*Parallel k-Nearest Neighbor*",[Online], Available: http://alitarhini.wordpress.com/2011/02/26/parallel-k-nearest-neighbor, 2011.

[5] Yiqun, C. , "*Parallel and Distributed Techniques in Biomedical Engineering*", M.Sc. Dissertation, Department of Electrical Computer Engineering, National University of Singapore, 2005.

[6] Zufrin, R. , "*Decision trees on parallel processors*", The International Joint Conference on Artificial Intelligence (IJCAI 1995) – Conference Proceedings, Montreal, Canada, August,1995.