Proceedings of the Twenty-Ninth International
Florida Artificial Intelligence Research Society Conference

Parallelizing Instance-Based Data Classifiers

Imad Rahal,[1] Emily Furst,[2] and Ramzi Haraty [3]

1 Department of Computer Science, College of Saint Benedict and Saint John's University, Collegeville, MN, USA
2 Department of Computer Science and Engineering, University of Washington, Seattle, WA, USA
3 Department of Computer Science and Mathematics, Lebanese American University, Beirut, Lebanon
irahal@csbsju.edu

Abstract

In the age of BigData, producing results quickly while operating over vast volumes of data has become a vital requirement for data mining and machine learning applications to a degree that traditional serial algorithms can no longer keep up with these constraints. This paper applies different forms of parallelization techniques to popular instance-based classifiers–namely, a special form of naive Bayes and *k*-nearest neighbors–in an attempt to compare performance and make broad conclusions applicable to instance-based classifiers. Overall, our experimental results strongly indicate that parallelism over test instances provides the most speedup in most cases compared to other forms of parallelism.

Background

Data-driven applications are witnessing an unacceptable degradation in performance due to rapidly increasing volumes of data. It is now common for applications to be expected to handle BigData which can be measured in the Terabytes, Petabytes and even Exabytes!

For many years now, computer architectures have followed Moore's Law; that is, the number of transistors that could be placed on a silicon chip doubled about every two years, allowing for continuous improvements in performance. In the past ten years, however, this exponential growth has slowed down due to three main factors referred to as the power wall, instruction-level parallelism (ILP) wall, and memory wall.

The power wall refers to the fact that since power consumption grows nonlinearly in comparison to clock speeds, computer architectures can no longer increase clock speed without requiring more power (thus generating more heat) than the chip can handle. Instruction-level parallelism refers to computing parallelism that is done implicitly by the hardware at runtime; hardware design has hit a wall in that computer architects cannot exploit any more parallelism automatically in this manner. Finally, memory wall is a trend in which off-chip memory access rates have grown much more slowly than processor speeds, meaning that performance bottlenecks due to memory latency and bandwidth are becoming more apparent (McCool, Robison and Reinders, 2012). As a result of these limits to computing performance, explicit parallelism where software applications are written to use available hardware parallelism has emerged as a necessary method for maintaining acceptable levels of performance.

Parallel computing, in contrast to traditional serial (or sequential) computing, allows programmers to specify computations that may be executed simultaneously (or in parallel). Threading is one technique for parallel computing; a thread refers to a separate flow of control which comes in two forms: hardware and software. A software thread can be thought of as a potential unit of parallelism, whereas a hardware thread is any hardware unit capable of executing a single software thread at a time. While the number of hardware threads is limited by the physical architecture, one may specify more software threads than there are hardware threads available. Any two threads that are running simultaneously must be independent of each other in order to avoid non-deterministic behavior as well as other issues such as deadlocks.

Two popular parallel patterns that are utilized in this work are the *map* and *reduce* patterns. The map pattern applies the same function to every element in an indexed collection of data. In other words, each iteration of a loop over a known number of data elements may be done in parallel. This implies that the computation done on the data element in each iteration is independent from all other iterations. The reduce pattern combines many data elements via an associative operator into one output data element. While each iteration is not independent, parallel models manage threads in such a way that only one thread may update the

resulting data element at a time (McCool, Robison and Reinders, 2012).

In this work, we use central processing unit (CPU) architecture parallelism (as opposed to graphics processing unit or GPU parallelism). In order to implement CPU parallelism, the OpenMP API (http://openmp.org/wp/) is utilized. OpenMP is a set of programming instructions and standards which use compiler directives, or pragmas, to alert the compiler to potential parallel blocks. Since CPUs can compute complex computations very quickly, CPU parallelism usually does well with parallelizing these complex computations.

The remainder of this paper is organized as follows: next, we provide a brief overview of other related works in the literature and then describe the algorithms studied, the proposed parallel implementations, the datasets utilized and the experimental setup. Experimental results are then reported and discussed followed by a conclusion and future direction.

Related Work

Our work focuses on two data mining classification algorithms, namely, *k*-nearest neighbors (*k*NN) and naive Bayes (NB) largely due to their popularity. Many studies in the literature have attempted to apply parallelism to both algorithms but usually in a very narrow fashion by focusing on certain application areas or datasets; we are not aware of studies that aim to make broad conclusions applicable to instance-based classifiers in general, like our work.

Numerous research works have applied parallelism to naive Bayes for the purpose of document classification. One such paper compared a GPU parallel implementation to a serial CPU implementation as well as a parallel implementation that uses four CPU cores (Viegas et al., 2013). Another work produced a five-time speedup using eight cores when naive Bayes was parallelized on the CPU using the Scala language (Peshterliev, 2012).

Some works have utilized cloud computing resources for data storage as well as the map-reduce patterns. One such paper used map-reduce for attribute parallelism but did not show significant time improvement from the serial version of their algorithm (Zhou, Wang H. and Wang W., 2012).

Similarly to naive Bayes, there has also been several implementations of *k*NN using both CPU and GPU parallelism. Garcia, Debreuve and Barlaud (2008) compared several GPU implementations and observed the best improvements using CUDA. Arefin et al. (2012) looked at an optimized version of the *k*NN algorithm implemented on a GPU using CUDA targeting parallelism when comparing and computing shortest distances. Garcia et al. (2010) is similar but proposed a CUDA implementation as well as a faster CUBLAS implementation although the focus was specifically on image processing. Nikan and Meshram (2014) looked at parallelizing the *k*NN algorithm on GPUs using the OpenCL API. Their proposed method of parallelism is similar to one of the implementations used in this study. They found sub-linear scale-up when using this method of parallelism which is in line with our results.

In addition to the above, some of the other related but fundamentally different works include Cesnovar et al. (2013) and Zhu et al. (2014) which applied parallelism to different image processing classification algorithms, Kato and Hosino (2010) which focused on distributed parallelism based on the architecture of the GPU, and Jin, Yang and Agrawal (2004) which studied parallel implementations to avoid race conditions–a special situation that occurs when the outcome of concurrent execution might be undesirable/incorrect unless a certain sequence of execution is guaranteed.

While research works referenced here all focus on parallelizing data mining classification algorithms in some way, shape or form, none shares the objective of our work which is to make broad conclusions applicable to instance-based classifiers (in general) that operate over datasets of various make-ups. For this purpose, we designed an experimental study to analyze the effect of the number of samples in the dataset, the number of attributes in the dataset, and the number of software threads on the different methods of parallelizing instance-based classifiers.

Parallel Implementations, Data and Experimental Setup

Our work focuses on two instance-based classifiers–meaning that no model is created from the training data beforehand. Note that although conditional probabilities are usually precomputed in naive Bayes, we opted to recompute the needed conditional probabilities for each test sample in order to justify the need for parallelism and to make comparisons between the two algorithms more meaningful. Note that such an instance-based naïve Bayes implementation may be useful in situations where data changes frequently and drastically (which is the case in data streams) mandating continuous retraining for standard naive Bayes implementations.

Both classifiers have three major loops in their algorithms: a loop over the test instances, a second loop over the training instances, and a third loop over the attributes. The first two loops can be clearly seen in the algorithms depicted in Figure 1. The first loop repeats the classification process (*k*NN or NB) for every test sample. The second loop in *k*NN computes the distance between the current test sample and every training sample in order to find the closest *k* training samples; in our instance-based NB,

the second loop finds the conditional probability for every attribute value in the current test sample given a class label in order to compute the overall conditional probability for that class label (given the attribute values in the current test sample). The third loop (i.e. over the attributes) is implicit when computing distances in *k*NN and probabilities in NB.

Based on these loops, we designed and implemented three parallel versions for each algorithm for a total of six implementations: *parallelism over the test instances* which treats every classification operation as one serial unit of execution and parallelizes the classification of the test samples over the available processors; *parallelism over the training instances* which parallelizes each classification operation by distributing the training samples' computations over the available processors; and *parallelism over the attributes* which parallelizes every distance computation in *k*NN and conditional probability computations in NB, both of which require looping over all existing attributes. For *k*NN, all implementations use the Euclidean distance measure.

```
kNN algorithm: input k
  loop over test samples t(v1,v2..vn):
    loop over training samples r:
      compute distance(t , r)
    end loop
    find the k r samples closest to t
    return majority class label c
  end loop
end algorithm


Instance-based NB algorithm:
  loop over test samples t(v1,v2..vn):
    loop over training samples r:
      compute probability (vi , c)
    end loop
    compute probability (c , t)
    return c with highest probability
  end loop
end NB algorithm
```

Figure 1: *Algorithms for k-nearest neighbors (kNN) and instance-based naive Bayes (NB).*

Data

In order to highlight the effect of increasing the number of attributes in the dataset on the performance of the different parallel implementations, we chose to utilize the following three datasets from the UCI Machine Learning Data Repository (http://archive.ics.uci.edu/ml) in our study: *Bank Marketing* dataset (referred to as DS1), *Musk* dataset (DS2), and *Internet Advertisement* dataset (DS3).

The datasets have 17 attributes, 168 attributes, and 1558 attributes, respectively, which increases the number of attributes by a factor of 10, roughly speaking. The Musk and Internet Advertisement datasets are numeric, binary classification datasets. The Bank Marketing dataset contains several categorical attributes which were converted into numeric. For each dataset, we produced two version: one containing one thousand randomly chosen instances and another one with ten thousand instances (using replication when needed). This amounts to six datasets in total (2 versions for each of the 3 original datasets). The resulting datasets were then divided into 75% training and 25% for testing.

Experimental Setup

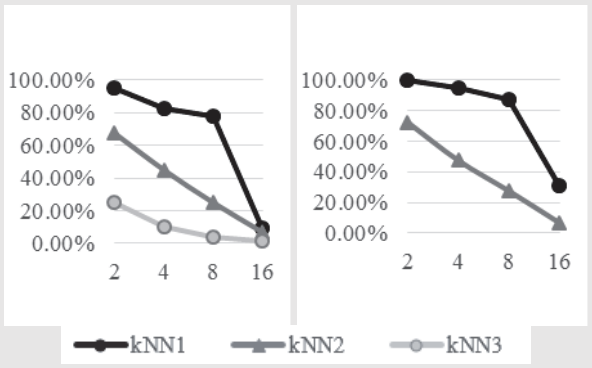
All programs were written in C++ using OpenMP and executed on an eight-node cluster called Melchior. Each node in Melchior is a dual Intel Xeon CPU E5-2420 Sandy Bridge 1.90GHz with 15MB L3 cache and six cores providing a total of 12 hardware thread states (2 thread states per core). Each node has 48GB memory.

In order to compare the various parallelization techniques, each implementation was run on each dataset version using 2, 4, 8, and 16 threads to study how performance scales as parallel resources increase. Each combination of parallel implementation, dataset version, and number of threads was run 10 times with the average execution time over the 10 runs being computed and reported. Speedup in execution time, S_p, was then computed as T_s/T_p where T_s is the runtime in serial and T_p is the parallel runtime using p threads; note that a value of S_p very close to p indicates linear speedup which is highly desirable. Instead of speedup, we report our results using efficiency which is a measure between 0 and 1 indicating how close speedup is to linear (with 1 being linear speedup or 100% efficient meaning that the resources used for parallelism are being fully utilized). Efficiency is defined as S_p/p or T_s/T_p*p. Note that for a given number of threads p, an efficiency value less than $1/p$ indicates performance worse than serial.

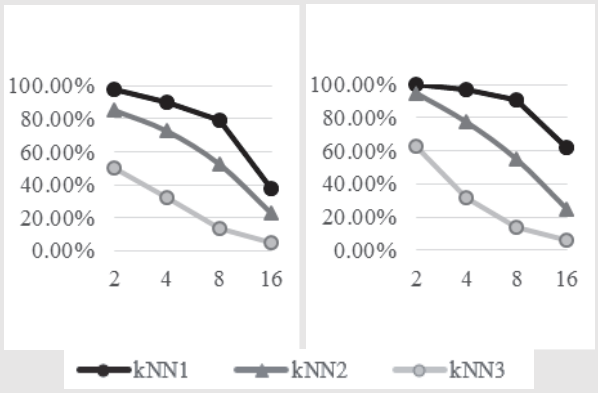
Results

Results depicted in Figure 2 and Figure 3 show efficiency (out of 100%) for each of the parallel implementations (given in the legend) for both *k*NN and NB, respectively, using a fixed number of threads (on the horizontal axis).

(a) Using 1000 (left) and 10000 (right) samples from DS1



(b) Using 1000 (left) and 10000 (right) samples from DS2



(c) Using 1000 (left) and 10000 (right) samples from DS3

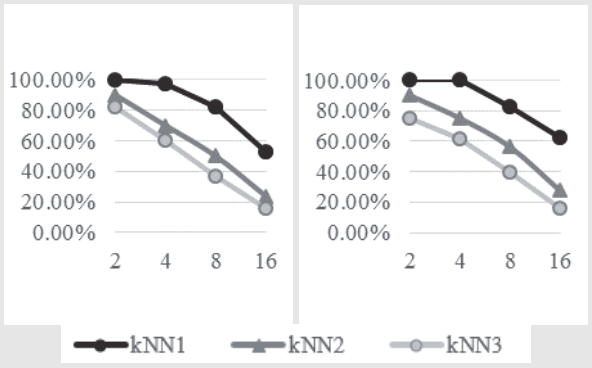
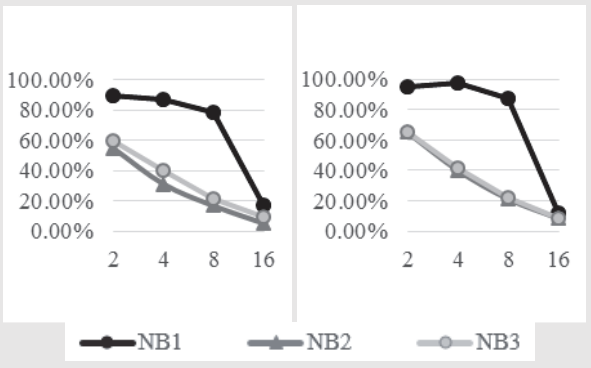
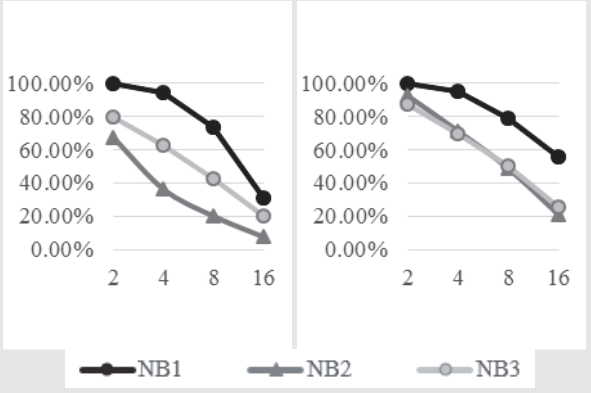


Figure 2: *Efficiency results for the kNN parallel implementations on all datasets while varying the number of software threads used.*

(a) Using 1000 (left) and 10000 (right) samples from DS1



(b) Using 1000 (left) and 10000 (right) samples from DS2



(c) Using 1000 (left) and 10000 (right) samples from DS3

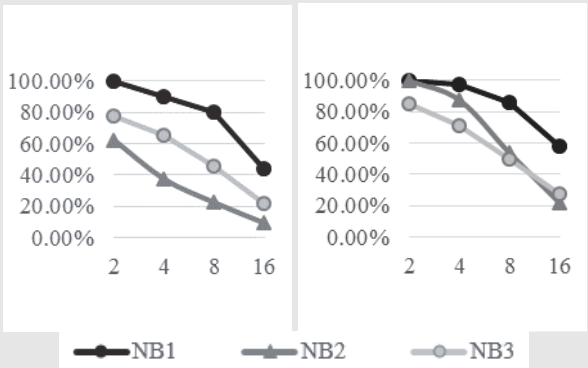


Figure 3: *Efficiency results for the NB parallel implementations on all datasets while varying the number of software threads used.*

Implementations are indexed from 1 to 3 where index 1 after the algorithm name (*k*NN or NB) refers to parallelism over the test instances, index 2 refers to parallelism over the training instances, and index 3 refers to parallelism over the attributes.

It is clear from Figure 2, which depicts efficiency results for the various *k*NN parallel implementations over the different dataset versions, that using up to 8 threads results in significant increase in speedup and efficiency, but going beyond that to 16 threads is neither efficient nor shows the expected speedup on all dataset versions but most notably on dataset DS1 which has the fewest attributes. This is most likely due to the fact that cluster nodes have only 12 thread states and the extra 4 threads in the 16 thread runs create too much overhead, slowing down the runtime. NB parallel implementations depicted in Figure 3 show similar behavior.

In terms of which implementation performs best, it is clear that while all efficiency curves show a decrease in value as we increase number of threads (which is expected due to parallelism overhead), parallelism over the test instances provides the best overall performance regardless of the number of threads used, the number of samples in the dataset, and the number of attributes in the dataset; at the same time, this implementation suffers the most when the number of requested threads exceeds the number of physical threads available. This applies to both algorithms, *k*NN and NB.

Parallelism over the training instances seems to produce the second best results for *k*NN especially when the number of attributes is relatively small (datasets DS1 and DS2). However, this favorable performance for this form of parallelism appears to degrade–thus approaching that of the parallelism over the number of attributes–once the number of attributes in the dataset increases significantly. Results for NB aren't as conclusive with most cases showing close efficiency for both types of parallelism when the number of attributes is relatively small. For dataset DS3, the efficiency seems to swing back and forth between the two implementations as the size of the dataset increases. Overall, parallelism over the number of attributes seems to have a slight edge in most cases for NB.

Conclusions and Future Work

Our implementations of *k*NN and NB indicate that parallelism over the test instances provides the most consistent levels of parallelism and tends to be very efficient especially when the number of software threads used does not exceed the available thread states.

Furthermore, as expected, parallelism over attributes only showed significant efficiency and speedup improvements as the number of attributes increases drastically. This is especially clear for the *k*NN implementation, although it is also still fairly obvious for the NB implementation. Preliminary results not reported herein using GPU parallelism via CUDA seem to support our CPU results as well.

While parallelism over the training instances performed fairly well for *k*NN, it still did not compare to the speedup and efficiency observed by parallelism over the test instances.

Finally, as the number of software threads requested exceeded the number of hardware thread states, significant decrease in performance was observed across the board for all parallel implementations regardless of the number of instances and/or attributes in the dataset.

In terms of future direction, we plan to build additional parallel implementations that utilize new types of parallelism as well as combinations of existing ones. We also aim to produce complete results for similar GPU parallelism using CUDA in order to validate our CPU conclusions.

References

Arefin, A.; Riveros, C.; Berretta, R.; and Moscato, P. 2012. GPU-FS-kNN: A Software Tool for Fast and Scalable kNN Computation Using GPUs. *PLoS ONE* 7(8): doi: 10.1371/ journal.pone.0044000.

Cesnovar, R.; Risojevic, V.; Babic, Z.; Dobravec, T.; and Bulic,P. 2013. A GPU Implementation of a Structural-Similarity-based Aerial-Image Classification. *The Journal of Supercomputing*, 65(2): 978-996.

Garcia, V.; Debreuve, E.; and Barlaud, M. 2008. Fast k Nearest Neighbor Search using GPU. In *Proceedings of the Computer Society Conference on Computer Vision and Pattern Recognition Workshops*, 1-6. Anchorage, USA: IEEE Press.

Garcia, V.; Debreuve, E.; Nielsen, F.; and Barlaud, M. 2010. k-Nearest Neighbor Search: Fast GPU-based Implementations and Application to High-Dimensional Feature Matching. In *Proceedings of the International Conference on Image Processing*, 3757-3760. Hong Kong, China: IEEE Press.

Jin, R.; Yang, G.; and Agrawal, G. 2004. Shared Memory Parallelization of Data Mining Algorithms: Techniques, Programming Interface, and Performance. *IEEE Transactions on Knowledge and Data Engineering*, 17(1): 71-89.

Kato, K., and Hosino, T. 2010. Solving k-Nearest Neighbor Problem on Multiple Graphics Processors. In *Proceedings of the International Conference on Cluster, Cloud and Grid Computing*, 769-773. Melbourne, Australia: IEEE Press.

McCool, M.; Robison, A.; and Reinders, J. 2012. *Structured Parallel Programming: Patterns for Efficient Computation*. San Francisco: Morgan Kaufmann Publishers.

Nikan, V., and Meshram, B. 2014. Parallel kNN on GPU Architecture using OpenCL. *International Journal of Research in Engineering and Technology*, 3(10): 367-372.

Peshterliev, S. 2012. Parallel Natural Language Processing Algorithms in SCALA. M.Sc. thesis, The Ecole Polytechnique Federale de Lausanne, Switzerland.

Viegas, F.; Andrade, G.; Almeida, J.; Ferreira, R.; Goncalves, M.; Ramos, G.; and Rocha, L. 2013.GPU-NB: A Fast CUDA-based Implementation of Naive Bayes. In *Proceedings of the International Symposium on Computer Architecture and High Performance Computing*, 168-175. Porto de Galinhas, Brazil: IEEE Press.

Zhou, L.; Wang, H.; and Wang, W. 2012. Parallel Implementation of Classification Algorithms Based on Cloud Computing Environment. *TELKOMNIKA Indonesian Journal of Electrical Engineering*, 10(5):1087-1092.

Zhu, L.; Jin, H.; Zheng, R.; and Feng, X. 2014. Effective Naive Bayes Nearest Neighbor based Image Classification on GPU. *The Journal of Supercomputing*, 68(2): 820-848.