Parallel k-Nearest Neighbors with MPI, OpenMP, and MPI-OpenMP Hybrid Programming using Text Data

Aaron Holt[1]

University of Colorado Boulder, Boulder, CO 80303, USA,
holtat@colorado.edu

Abstract. Parallel computing takes advantage of multiple processors to increase program performance. Many machine learning algorithms require computationally intense algorithms which can leverage the additional processors. This paper will investigate the performance change as well as any changed in accuracy after parallelizing the k-nearest neighbors (kNN) algorithm using three different methods for different processor counts in C. The kNN algorithm will be parallelized with a fixed feature set taken from text data using a straight MPI implementation, an OpenMP only implementation, and a hybrid MPI-OpenMP implementation. After comparing results from the three implementations, it was found that the pure MPI implementation was the fastest for all processor counts. The hybrid implementation was the next fastest, and the OpenMP version was the slowest overall. There was no change in accuracy for any of the implementations.

Keywords: C, k-Nearest Neighbors, Machine Learning, MPI, OpenMP

1 Introduction

1.1 The k-Nearest Neighbor Algorithm

kNN is a widely used algorithm in data mining and machine learning. It is used for clustering as well as classification and is known for being a simple algorithm with a relatively low error rate [5]. kNN is an instance-based learner which classifies a given point based on nearby points. The general idea behind kNN is that if a given points in a feature space have similar feature, than they will all be close within this space. One of the benefits of kNN is that the training required by other algorithms isn't necessary. A featurized set of training data with classification labels is all that is needed. Every point p of the training set must exist in the same n-dimensional space and be expressed as in equation 1, where the x values represent individual features. The same requirements must also be met for the test set.

$$p = (x_0, x_1, ..., x_{n-1}, x_n) \tag{1}$$

After all the training and test data has been featurized, a distance function can be used to calculate the distance between two points. Euclidean distance is typically used and was chosen for this paper. Thus the distance between points p and q is seen in equation 2

$$dist(p, q) = \sqrt[2]{\sum_{i=1}^{n} (p_i - q_i)^2} \tag{2}$$

In order to classify point q from the test set, it must be compared to every point p in the training set. The k closest points to q determine its class. Specifically, the most common class within the k closest points is what q will be classified as. In the event of a tie, the class with the closest point overall is the winner. Other tie-breaking measures exist, but they typically have little impact on the algorithm as a whole as ties are usually rare.

1.2 Machine Learning with Text Data

One of the most challenging parts of any machine learning task is feature engineering. Turning a real world problem into meaningful numbers requires a solid understanding of the problem at hand. A good baseline approach for most text-based problems is called Bag-of-Words [3]. Bag-of-Words takes every word of a text and turns it into a feature. Another common procedure is to remove stopwords, or words such as 'and', 'the', 'this', etc. This typically increases the accuracy of the Bag-of-Words approach while also reducing the number of dimensions. In order to turn the text into numeric data of the form seen in 1, every feature must have a unique dimension (feature number) associated with it. Common techniques include hash functions and and inserting the features into a tree and performing a traversal to number them. For this paper, each word from training set was inserted into a binary search tree and then numbered via an in order traversal.

2 Related Work

2.1 Parallel kNN

Machine learning is a computationally intense field and as such there is a strong interest optimizing performance. kNN is an embarrassingly parallel algorithm and as such has been parallelized using various techniques. Most of the work surrounding kNN is focused on GPU applications such as CUDA as GPU's are specialized to handle compute-intensive parallel applications. Compared to CPU based applications, GPU applications can significantly increase performance [5]. Other approaches include modifying the kNN algorithm itself to improve accuracy. The Approximate Nearest Neighbor (ANN) trades accuracy for an increase in performance. Others have tried breaking the algorithm into modular sections to allow for improved accuracy and increased performance (via parallelism) [6].

2.2 Comparing OpenMP and MPI

With the increasing variety of tools available for parallelism, a significant amount of work has been done comparing each tools performance for various applications. One relevant study compared the performance of MPI to OpenMP-MPI hybrid programs on the NAS benchmarks. The overall performance depended mostly on the level of shared memory parallelism, communication patterns, memory access patterns, and computer architecture. For most of the tests performed, the pure MPI applications outperformed the OpenMP-MPI hybrid ones. The exceptions were for benchmarks where large message sizes dominated the communication time and slowed the MPI application [8].

3 Dataset and Problem Definition

The dataset chosen is a collection of 41,500 Quiz Bowl questions. Quiz Bowl is an academic trivia competition that tests players on a wide variety of subjects. In Quiz Bowl, players are given a category and a question and have to find the answer. As this paper focuses on scalability and performance rather than accuracy, the goal is to guess the category based on the question.

4 Parallelizing kNN

4.1 Data Structures

The first objective of the code was to parse the csv file containing the Quiz Bowl questions and use the Bag-of-Words approach to generate features. An array of struct *data*s as seen below was constructed to store the featurized data in text form. In this struct, the char **question pointed to an array for char *'s which contained individual words.

Listing 1.1: Data structure to store featurized text data

```
//stores data in text form
struct data{
        int example_num;
        char **question;
        char *cat;
};
```

The second objective of the code was to collect every feature into a data structure. With over 40,000 questions each containing roughly 120 words, the total feature count was approximately 150,000. Thus using an array to store each feature and make sure it is unique was unfeasible. A binary search tree was used to store the features, ensure each feature was unique, and assign a unique number to each feature. A for loop over the featurized text data was used to add all the features to the binary search tree. After, an in order traversal was used to assign each feature a unique number. After the feature tree was finished, a for loop over the text data was used to convert the text data into numeric data. The numeric data was stored in an array of *numeric_data* structs as seen below:

Listing 1.2: Data structures to store sparse numeric feature data

```
//stores data in numeric form
struct numeric_data{
        int example_num;
    int total_features; //number of features for given example
        struct feature_count *array_of_features;
        int cat;
};

//data is sparse (each example has ~100/150,000 points)
//using 'numeric_data' and 'feature_count' structs to store sparse data
struct feature_count{
        unsigned int feature_num;
        unsigned char count;
};
```

4.2 Parallel kNN

After the data was in numeric form, the kNN algorithm could be used to classify new questions. Three parallel versions were coded, each having the option to run in sequentially. As mentioned in the abstract, there was a strictly MPI version, a strictly OpenMP version, and an OpenMP-MPI hybrid version.

MPI In order to parallelize the MPI only portion, the featurized numeric data was copied onto each MPI process. Every process was assigned a range of points to classify. After every point was classified, an MPI_Reduce was used to collect the results onto process 0 such that they could be written to an outfile. The code was structured as follows, where the *range* variable determined which points each process classified.

Listing 1.3: OpenMP code structure

```
//Loop over data, classify points
for (kk=rank*range; kk<(rank+1)*range; kk++){
    //calc distance to neighbors
    for (ii=0; ii<total_examples-1; ii++){
        ...
    }
}
MPI_Reduce() //Reduce correctly answered
MPI_Reduce() //Reduce total answered
```

OpenMP For the OpenMP version, the featurized numeric data was not copied to each thread. Instead the numeric data was declared shared such that every thread created would have access to it. This approach decreases the overall amount of memory used, but slows down the performance. The number of threads was manually specified by a command line input. Reduction variable were declared to gather results at the end of the parallel section. The code was structured as follows:

Listing 1.4: OpenMP code structure

```
#pragma omp parallel
//OpenMP clauses here
{
        //loop over data, classify points
        #pragma omp for
    for (kk=0; kk<total_examples; kk++){
        //calc distance to neighbors
        for (ii=0; ii<total_examples-1; ii++){
                ...
        }
    }
}
```

OpenMP-MPI Hybrid The hybrid version had the numeric data copied to each MPI process, but then had each OpenMP thread spawned from that process share the variable. Each MPI process had a specified range of points to classify, which OpenMP then broke down further. The number of OpenMP threads was specified manually via a command line argument. The number of OpenMP threads per MPI process is equal to the total number of MPI processes. Thus if 3 MPI processes were created, each MPI process would then have 3 OpenMP threads for a total of 9 tasks. The code was structured as follows:

Listing 1.5: Hybrid code structure

```
#pragma omp parallel
//OpenMP clauses here
{
        //loop over data, classify points
        #pragma omp for
    for (kk=rank*range; kk<(rank+1)*range; kk++){
            //calc distance to neighbors
            for (ii=0; ii<total_examples-1; ii++){
                    ...
            }
        }
}
```

4.3 Accuracy and Timing Methodology

As is typical with most machine learning problems, a portion of the data is withheld for testing. In this case 90% of the data was used for training and 10% was used to test on. The correct answer for the 10% was known beforehand and used to calculate the accuracy of the algorithm.

In order to verify correctness of the each version, the results were compared to the sequential version to ensure that they matched. This was done for each k number used as well as every processor count used for each version. Timing for all tests was done on the Janus supercomputer.

MPI_Wtime was used to time the MPI and hybrid code versions. The resolution of *MPI_Wtime* was found using the *MPI_Wtick* function and found to be 0.000001 seconds. This was significantly less time than shortest test run, which was approximately 0.02 seconds. For the OpenMP only version, the *gettimeofday* function was used which has a resolution of 0.00001 seconds, which is still significantly shorter than any test run.

Several sets of timing tests were run to test the scalability of the parallelized kNN algorithm. Each test ran the full algorithm 11 times total. The first iteration was not timed and instead used as a burn in run. The next 10 iterations were timed and averaged. The points on the following graphs are all averages of the 10 timed iterations. Error bars on all graphs represent the 95% confidence interval for the 10 points. The error was so small for most points that the error bars are not visible. The following is a list of the scalability tests run:

1. Fix the number of tasks and number of questions used, increase k.
2. Fix the number of questions used and k, increase the number of tasks.
3. Fix the number of tasks and k, increase the number of questions used.

It should be noted that the majority of the tests were run with 10,000/41,500 questions to save on core hours.

5 Results

5.1 Varying k

Figure 1 shows accuracy versus k on the left and the runtime versus k on the right. The accuracy increased with k until its peak at k=6 and an accuracy of almost 68%. After this point it slowly dropped off. Typically an increase in k causes an increase accuracy as more points get averaged and outliers become less impactfull. In this case k started to decrease at around 10 due to the dataset size. Several categories had significantly fewer questions associated with them in the training set. Thus as k increased, it became more and more likely that points from other categories would be close and so more points were misclassified. The runtime should increase linearly with an increase in k. This was the case for all three implementations. There was an exception for between k=1 and k=4 for the OpenMP version. There was no clear reason why this occurs.

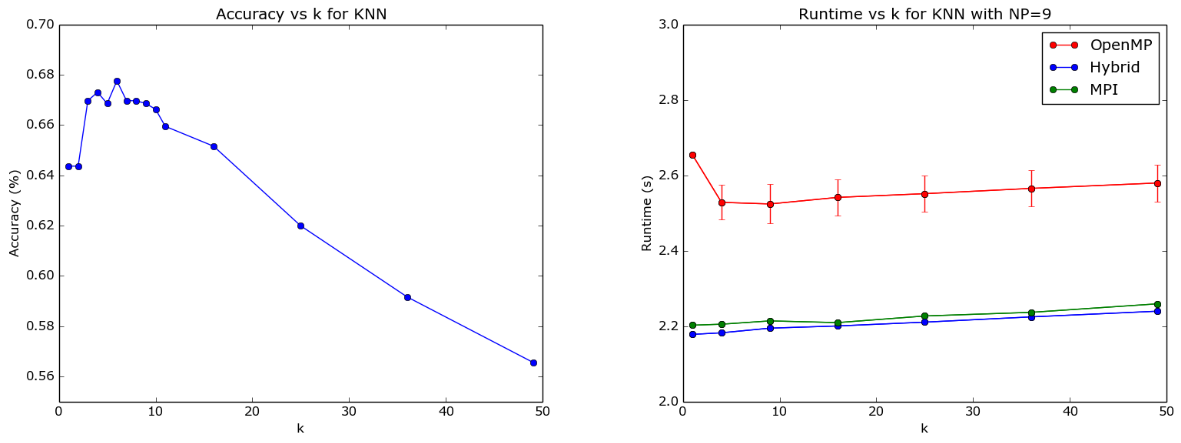

Fig. 1: Runtime and Accuracy versus k with 10,000 questions used

5.2 Varying number of tasks

Figure 2 shows the runtime versus the number of tasks for each implementation. It should be noted that the strictly OpenMP version could only be run on one node. Each node on Janus has 12 processors. This means that any thread count higher than 12 will no longer have a processor devoted to it.

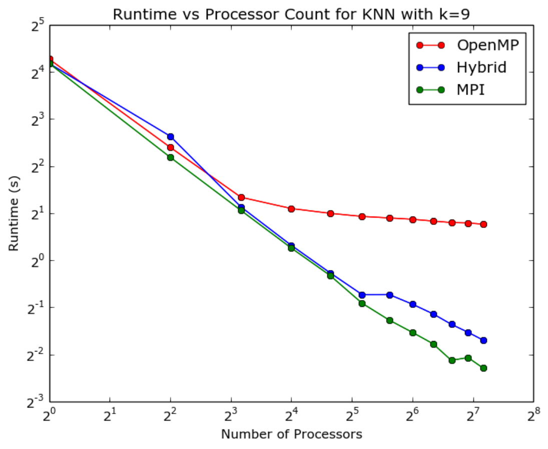

Fig. 2: Runtime vs number of tasks for 10,000 questions

As can be seen in Figure 2, the MPI only implementation has the best overall performance. The hybrid version was close to the MPI version until 36 processors at which point it began to drop off. The likely reason for the dropoff in performance for the hybrid version is the OpenMP overhead as well as the shared variables used. As the number of threads increases, more threads have to share the numeric data, slowing the runtime. The OpenMP only version dropped off quickly after 12 processors as expected.

Figure 3 shows the speedup and efficiency for the same test.

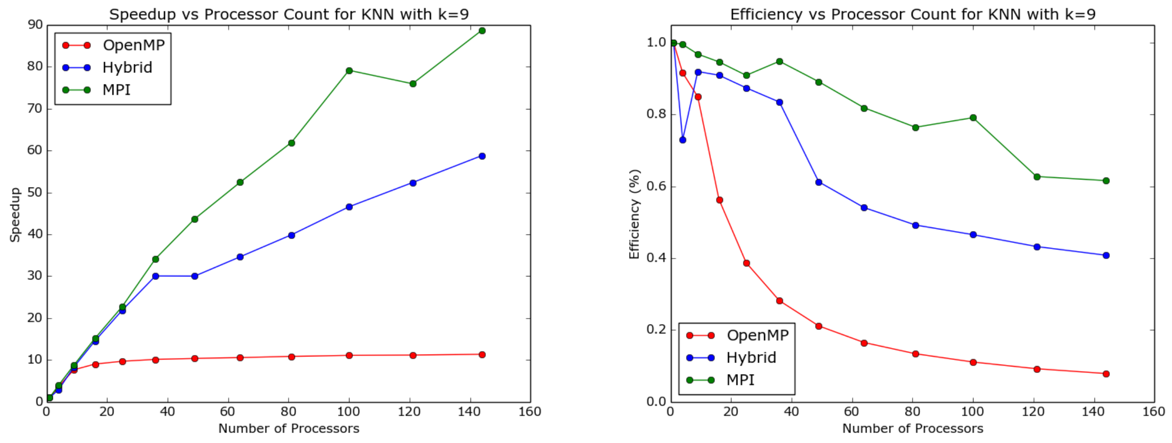

Fig. 3: Speedup and efficiency vs number of tasks for 10,000 questions used

As can be seen in Figure 3, the MPI implementation had the best speedup and efficiency in all cases. The speedup increased almost linearly with the number of processes and the efficiency decreased linearly until it reached 62% at 144 processors. The OpenMP efficiency and speedup dropped sharply after 12 threads. Surprisingly, some speedup was achieved at high thread counts. The hybrid version had near the efficiency and speedup of the MPI version until 36 processors, at which point it dropped off. As mentioned before this is likely due to the shared variable. There is one noteable outlier in the efficiency plot for the hybrid version with 4 tasks. It has a significantly lower efficiency than the points around it. This is discussed further in the profiling section.

5.3 Varying amount of data used

Figure 4 shows the increase in runtime as the dataset is increased. The runtime increases exponen-

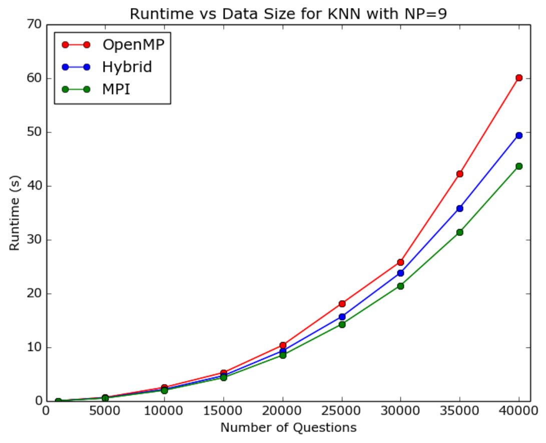

Fig. 4: Runtime vs Number of Tasks for 10,000 questions

tially with the number of questions. This is because as the number of questions increases, there is also an increased number of questions to classify as well as an increased number of questions to compare against.

5.4 Profiling

Initial profiling was done using Allinea for each version with k=9, number of tasks=9, and 10,000 questions. The results are shown below in tables 1 and 2.

	MPI (%)	OpenMP (%)	Hybrid (%)
Compute	85.5	94.9	87.6
MPI Call	14.5	0	10.8
OpenMP	0	5.1	1.6

Table 1: Kernel split

Table 1 shows what percentage of time each program spent in computation and in communication. It should be noted that the MPI call time is high as some of it comes from a barrier at the end of the program before a write to a file. As kNN is an embarrassingly parallel algorithm, it makes sense that the majority of the time spent is during compute time.

	MPI (%)	OpenMP (%)	Hybrid (%)
Train	5	2	4
Test	83	94.9	87.6
get_distance	78	93.8	84.8

Table 2: Time spent on training and testing

Table 2 shows where in the computation time was spent in the program. Almost no time was spent in training (reading in the data, featurizing and making it numeric). The majority of the time was spent classifying points. Almost all the time for every implementation was spent in the *get_distance* function. This function calculates the distance between two points. This function could have been improved by changing the data structure which held the numeric data. The current data structure is an array. The data is stored in a sparse array. Thus when calculating distance the *get_distance* function has to check if two questions have the same features. This takes linear time when the features are in an array. If the features were in a tree, the same process could take logarithmic time. With roughly 120 features per question, this could reduce the time taken by a factor of 7.

The main outlier in the results was from the hybrid efficiency plot where the run with 2 MPI processes each with 2 OpenMP threads was significantly less efficient than nearby points. Valgrind's cachegrind was run in an attempt to see if memory access played a part in this outlier. The outlying point had an L1 miss rate of 0.5% versus 0.4% of nearby points and a last level miss rate of 0.4% versus 0.0% of nearby points. This likely doesn't account for the entire drop in efficiency, but accounts for some of it. It is unclear why the cache performance changed for that particular run.

6 Further Work

The code used for this project could be further optimized to get a better idea of how scalable the kNN algorithm really is. Fixing the *get_distance* function would be a good first step. The next step would be to profile the code further and optimize memory access. As the hybrid and OpenMP versions used shared variables, it was rather unfair to compare them to a program with no shared variables. Rewriting the OpenMP and hybrid codes such that each thread has its own copy of the data would make for a better performance comparison between the three implementations. Finally, many machine learning algorithms use the same data input. With code that already featurizes text data, other machine learning algorithms could be parallelized and compared.

7 Conclusion

The k-Nearest Neighbors is a relatively simple yet effective machine learning algorithm. As a computationally intense algorithm, parallelism can dramatically reduce runtime. Three different implementations of parallel kNN were compared. The MPI only version had the best performance, followed by the hybrid and OpenMP versions. The hybrid and OpenMP version used shared variables which likely slowed their performance.

References

1. Georgiana Ifrim, Gkhan Bakir, and Gerhard Weikum. 2008. Fast logistic regression for text categorization with variable-length n-grams. In Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining (KDD '08). ACM, New York, NY, USA, 354-362.
2. Yong Zhuang, Wei-Sheng Chin, Yu-Chin Juan, and Chih-Jen Lin. 2014. Distributed Newton Methods for Regularized Logistic Regression. National Taiwan University, Taipai, Taiwan.
3. Daniel Jurasfky, and James Martin. Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition. Pearson, 2008.
4. Peter Lubell-Doughtie and Jon Sondag. Practical Distributed Classification using the Alternating Direction Method of Multipliers Algorithm. Intent Media, New York, USA.
5. Zhao, Lei, and Quansheng Kuang. A Practical GPU Based KNN Algorithm. Proc. of Second Symposium International Computer Science and Computational Technology, China, Huangshan. N.p.: n.p., n.d. 151+. Print.
6. Zhao, Hai, and Bao-Liang Lu. A Modular K-Nearest Neighbor Classification Method for Massively Parallel Text Categorization. Department of Computer Science and Engineering, Shanghai Jiao Tong University, n.d. Web.
7. Jost, Gabriele, Haoqiang Jin, Dieter An Mey, and Ferhat F. Hatay. "Comparing the OpenMP, MPI, and Hybrid Programming Paradigms on an SMP Cluster1." NAS (2003): n. pag. Web.
8. Cappello, Franck, and Daniel Etiemble. MPI versus MPI+OpenMP on the IBM SP for the NAS Benchmarks. LRI, Universite Paris-Sud, n.d. Web.