CSC 410 Final
Daniel Albl

Overview:

This program demonstrates a parallel implimentation of the K nearest neighbors algorithm, using the "Iris" dataset. This dataset contains 4 attributes for each iris: sepal length, sepal width, petal length, and petal width, and whether they are classified as "setosa", "viginica", or "versicolor".

The implementation of the knn algorithm used in this program consists of 3 steps:
1) Calculate the Euclidean distance from the iris to be classified and each of the irises in the dataset.
2) Sort the dataset according to this distance.
3) Classify the iris based on the mode of the class labels of the first K training samples.

The data is also standardized, meaning subtracting the mean from each attribute and dividing by their standard deviation. This ensures each attribute will have an equal impact on the distance measure.

Partitioning:

The first step can be partitioned into the calculation of the distance to a single training iris. From the data perspective, this means partitioning the dataset into individual data points.

The sorting algorithm I used for the second step was merge sort. This can be broken into the merging of two sorted halves of a sub-array. The merge step could be further partitioned down to each array index, but this requires each index to do an additional log(n) search of the other array. Because this program uses openMP which uses relatively few threads, I decided against partitioning the merge step.

The final step of finding the mode is a O(k) operation, and because k is generally small and finding the mode requires parallel reduction, I decided against parallelizing the final step.

Communication:

The distance calculations do not require any communication. For merge sort, although parallel processes don't communicate, processes do create sub-processes, which must "communicate" to the parent process that they have completed.

Agglomeration:

For calculating distance, the agglomeration does not need to minimize communication and is only for reducing the number of processes to what is available on the hardware. Combining predetermined continuous chunks of the array (omp's static scheduling) yielded the best performance in practice. This is likely because each calculation should take roughly the same time making dynamic scheduling a waste of overhead, and continuous chunks of memory are usually better for the cache.

For merge sort, tasks can be combined when the size of the array is below a certain threshold. In practice, I found that making tasks unable to create child tasks when their parent task's array was <= 8 was the most efficient.

Mapping:

This implementation did not require mapping as it was done using shared memory.

Implementation:

The program uses 2 structs: “Iris”, which holds the attributes and class for an iris data point, and “Idx”, which holds the distance from a point to the test iris, and the index of the point in the array of training irises. There are global variables for the test iris, the array of training points, the array of “Idx” structs, and arrays for the means and standard deviations for the attributes.

The program begins with the function “loadData”, which reads the training data from the file “iris.data”. Next, the function “norm” finds the means and standard deviations and uses these to standardize the data. Then “makeTest” uses command-line arguments to set the attributes for the test iris, after standardizing them. “Knn” does the actual work of the program, calculating the distances, sorting, and then returning the mode of the k nearest neighbors. This uses the helper functions “dist”, “mode”, and “argMax3”. The sorting is done with the “mergesort” functions, which recursively calls itself and the “merge” function. Each recursive call of “mergesort” uses partitions of the index-distances array and a buffer array for merging.

Performance:

Parallelizing the calculation of distances resulted in an obvious speed up. For a dataset of size 10^7, the serial implementation averaged 0.2325 seconds, while the parallel version averaged 0.0694. This gives a speedup of 3.35x and a parallel efficiency of 0.419%, as it was done using 8 threads. Merge sort, however, was actually taking longer when parallelized and I unfortunately ran out of time to figure it out.

How To Run:

```
$ make
$ ./main <size of rand dataset>  (for timing code)
$ ./main <sepal length> <sepal width> <petal length> <petal width> (for classification)
```