CS 447/547: Parallel *k*-NN

Due March 13, 11:59 PM.

(This document was last modified on Monday, February 24, 2020 at 11:57:03 AM.)

The topic of the assignment is implement a parallel *k*-NN program. The method of parallelization should be threads. You may use C++ (recommended) or C, or any mix of the two. Other languages may be allowed. If you are interested, you may contact the instructor.

The *k*-NN problem is to find the *k* nearest neighbors of a given query point. When used in data science, the assumption is that the neighbors of a point will have similar characteristics to the query point. For example, suppose you are trying to predict the favorite movie of a person. It would be reasonable to assume that if Joe is similar to Mary in other ways, they might have the same favorite movie.

The *k*-NN problem is commonly solved using *k*-d trees. A *k*-d tree is like a binary search tree, except that each node represents a split along a different dimension. That is, each node corresponds to a particular dimension, and the split is along that dimension. Thus, each node can be thought of as representing a hyperplane. Data points to the negative side of the hyperplane are in the left subtree, while data points to the positive side of the hyperplane are in the right subtree. The dimensions typically alternate in a round-robin fashion as the depth increases. Thus, the root node (zeroeth level) might split along the *x* dimension, first level nodes along the *y* dimension, second level nodes along the *z* dimension, third level nodes along the *x* dimension, etc.

As with binary search trees, balance is an issue. If the tree is unbalanced, search times will increase. *k*-d trees can be balanced by choosing the median point (in the appropriate dimension) as the split value. The median point can be found exactly by sorting the points, or can be approximated by randomly choosing a subset of the points and sorting those.

Searching a *k*-d tree is relatively straightforward. Given a query point, the tree is first descended as if inserting the point. Once a leaf node is reached, the tree is followed upward. At each node, the distance to the hyperplane is compared against the current nearest neighbors. If needed, the other branch is then descended.

Requirements

The program take four arguments on the command line.

> `./k-nn` *n_cores training_file query_file result_file*

The number of cores is that have been allocated for this execution is given by *n_cores*. You may use this to better determine how many threads to create. (You can also query this programmatically by calling `sched_getaffinity()`. The training points are given in *training_file*. The queries are given in *query_file*. The results should be written to *result_file*. To prevent overwriting an important file, you may wish to confirm that it does not already exist. To minimize I/O processing time, the files are in binary (native byte order).

Training File Format

The header of the training file is as below.

Field	Size in Bytes	Data Type	Notes

Field	Size in Bytes	Data Type	Notes
File type string	8	Array of chars, padded out with zeroes	The type string for training files is `"TRAINING"`. This is exactly 8 chars, so there is no 0 byte at the end.
Training file ID	8	Unsigned, 64-bit integer	A randomly-generated number that can be used to uniquely identify files.
Number of points	8	Unsigned, 64-bit integer	
Number of dimensions	8	Unsigned 64-bit integer	

After the header are the points, given as a sequence of 32-bit floating-point numbers.

The training file ID is a opaque, unique ID. It purpose is to help correlate training files with result files.

Query File Format

The header of the query file is:

Field	Size in Bytes	Data Type	Notes
File type string	8	Array of chars, padded out with zeroes	The type string for query files is `"QUERY"`.
Query file ID	8	Unsigned, 64-bit integer	A randomly-generated number that uniquely identifies files.
Number of queries	8	Unsigned, 64-bit integer	
Number of dimensions	8	Unsigned 64-bit integer	
Number of neighbors to return for each query	8	Unsigned 64-bit integer	

After the header are the queries, given as a sequence of 32-bit floating-point numbers.

The query file ID is a opaque, unique ID. It purpose is to help correlate query files with result files.

Results File Format

The header of the results file is:

Field	Size in Bytes	Data Type	Notes
File type string	8	Array of chars, padded out with zeroes	The type string for query files is `"RESULT"`.
Training file ID	8	Unsigned, 64-bit integer	The ID of the training file used to generate this results file.
Query file ID	8	Unsigned, 64-bit integer	The ID of the query file used to generate this results file.
Result file ID	8	Unsigned, 64-bit integer	A randomly-generated number that uniquely identifies this file.

Field	Size in Bytes	Data Type	Notes
Number of queries	8	Unsigned, 64-bit integer	
Number of dimensions	8	Unsigned 64-bit integer	
Number of neighbors returned for each query	8	Unsigned 64-bit integer	

After the header are the results, given as a sequence of 32-bit floating-point numbers.

The training file ID must match the ID given in the training file. The query file ID must match the ID given in the query file. The result file ID will be generated by your program. Generate a random number by reading from `/dev/urandom`.

Timing Outputs

The program must print the total time (wall clock) to build the tree and the total time to execute the queries.

Sample Code

Here is a sample code that you can use to get a sense of the scale that you should be aiming for. It is a program that will create a file of random doubles, and another program that will read them in and put them in a BST. With 20,000,000 doubles, it takes about 25 seconds and less than 1 GB of memory. The sample code also shows how you can use `mmap()` to simplify your code.

Extra Credit (50 points)

For extra credit, also use parallelization to improve the speed of single queries. For that, you must read the query from a named pipe.