Assignment 5 (Adapted from the project by Dan Grossman)

Introduction

In this project, you will process some data from the 2010 census to efficiently answer certain queries about population density. These queries will ask for the population in some rectangular area of the country. The input consists of "only" around 220,000 data points, so any desktop computer has plenty of memory.

Overview of what your program will do

The file CenPop2010.txt contains real data published by the U.S. Census Bureau. The data divides the U.S. into 220,333 geographic areas called "census-block-groups" and reports for each such group the population in 2010 and the latitude/longitude of the group. It actually reports the average latitude/longitude of the people in the group, but that will not concern us: just assume everyone in the group lived on top of each other at this single point.

Given this data, we can imagine the entire U.S. as a giant rectangle bounded by the minimum and maximum latitude/longitude of all the census-block-groups. Most of this rectangle will not have any population:

- The rectangle includes all of Alaska, Hawaii, and Puerto Rico and therefore, since it is a rectangle, a lot of ocean and Canada that have no U.S. population.
- The continental U.S. is not a rectangle. For example, Maine is well East of Florida, adding more ocean.

Note that the code we provide you reads in the input data and changes the latitude for each census group. That is because the Earth is spherical but our grid is rectangular. Our code uses the Mercator Projection to map a portion of a sphere onto a rectangle. It stretches latitudes more as you move North. You do not have to understand this except to know that the latitudes you will compute with are not the latitudes in the input file.

Your program will first process the data to find the four corners of the rectangle containing the United States. Some versions of the program will then further preprocess the data to build a data structure that can efficiently answer the queries described above. The program will then prompt the user for such queries and answer them until the user chooses to quit.

More details on how your program should work

The first three command-line arguments to your program will be:

- The file containing the input data
- Two integers x and y describing the size of a grid (a two-dimensional array) that is used to express population queries

Suppose the values for x and y are 100 and 50. That would mean we want to think of the rectangle containing the entire U.S. as being a grid with 100 columns (the x-axis) numbered 1 through 100 from West to East and 50 rows (the y-axis) numbered 1 through 50 from South to North. (Note we choose to be "user friendly" by not using zero-based indexing.) So the grid

would have 5000 little rectangles in it. Larger x and y will let us answer queries more precisely but will require more time and/or space.

A query describes a rectangle within the U.S. using the grid. It is simply four numbers:

- The Western-most column that is part of the rectangle; error if this is less than 1 or greater than x.
- The Southern-most row that is part of the rectangle; error if this is less than 1 or greater than y.
- The Eastern-most column that is part of the rectangle; error if this is less than the Western-most column (equal is okay) or greater than x.
- The Northern-most row that is part of the rectangle; error if this is less than the Southern-most column (equal is okay) or greater than y.

Your program should read these four numbers. Any illegal input (i.e., not 4 integers on one line) indicates the user is done and the program should end. Otherwise, you should output two numbers:

- The total population in the queried rectangle
- The percentage of the U.S. population in the queried rectangle, rounded to two decimal digits, e.g., 37.22%

You should then repeat for another query.

DO NOT CHANGE THE PROVIDED I/O FORMAT AS IT WILL BE USED FOR TESTING.

To implement your program, you will need to determine within which grid rectangle each census-block-group lies. This requires computing the minimum and maximum latitude and longitude over all the census-block-groups. Note that smaller latitudes are farther South and smaller longitudes are farther West. Also note all longitudes are negative, but this should not cause any problems.
In the unlikely case that a census-block-group falls exactly on the border of more than one grid position, tie-break by assigning it to the North and/or East as needed.

Six Different Implementations
You will implement 6versions of your program.

Version 1: Simple and Sequential
Before processing any queries, process the data to find the four corners of the U.S. rectangle using a sequential O(n) algorithm where n is the number of census-block-groups. Then for each query do another sequential O(n) traversal to answer the query (determining for each census-block-group whether or not it is in the query rectangle). The simplest and most reusable approach for each census-block-group is probably to first compute what grid position it is in and then see if this grid position is in the query rectangle.

Version 2: Simple and Parallel

This version is the same as version 1 except both the initial corner-finding and the traversal for each query should use goroutines. The work will remain O(n), but the span should lower to O(log n). Finding the corners should require only one data traversal, and each query should require only one additional data traversal.

Remember that you “MUST” have a sequential cut-off after which the sequential

Version 3: Smarter and Sequential

This version will, like version 1, not use any parallelism, but it will perform additional preprocessing so that each query can be answered in O(1) time. This involves two additional steps:

1. First create a grid of size x*y (use an array of arrays) where each element is an int that will hold the total population for that grid position. Recall x and y are the command-line arguments for the grid size. Compute the grid using a single traversal over the input data.
2. Now modify the grid so that instead of each grid element holding the total for that position, it instead holds the total for all positions that are neither farther East nor farther South. In other words, grid element g stores the total population in the rectangle whose upper-left is the North-West corner of the country and the lower-right corner is g. This can be done in time O(x*y) but you need to be careful about the order you process the elements. Keep reading...

For example, suppose after step 1 we have this grid:
0 11 1 9
1 7 4 3
2 2 0 0
9 1 1 1

Then step 2 would update the grid to be:
0 11 12 21
1 19 24 36
3 23 28 40
12 33 39 52

There is an arithmetic trick to completing the second step in a single pass over the grid. Suppose our grid positions are labeled starting from (1,1) in the bottom-left corner. (You can implement it differently, but this is how queries are given.) So our grid is:
(1,4) (2,4) (3,4) (4,4)
(1,3) (2,3) (3,3) (4,3)
(1,2) (2,2) (3,2) (4,2)
(1,1) (2,1) (3,1) (4,1)

Now, using standard array notation, notice that after step 2, for any element not on the left or top edge: grid[i][j] := orig+grid[i-1][j]+grid[i][j+1]-grid[i-1][j+1] where orig is grid[i][j] after step 1. So you can do all of step 2 in O(x*y) by simply proceeding one row at a time top to bottom -- or one column at a time from left to right, or any number of other ways. The key is that you update (i-1 , j), (i , j+1) and (i-1 , j+1) before (i , j).

Given this unusual grid, we can use a similar trick to answer queries in O(1) time. Remember a query gives us the corners of the query rectangle. In our example above, suppose the query rectangle has corners (3,3), (4,3), (3,2), and (4,2). The initial grid would give us the answer 7, but we would have to do work proportional to the size of the query rectangle (small in this case, potentially large in general). After the second step, we can instead get 7 as 40 - 21 - 23 + 11. In general, the trick is to:

- Take the value in the bottom-right corner of the query rectangle.
- Subtract the value just above the top-right corner of the query rectangle (or 0 if that is outside the grid).
- Subtract the value just left of the bottom-left corner of the query rectangle (or 0 if that is outside the grid).
- Add the value just above and to the left of the upper-left corner of the query rectangle (or 0 if that is outside the grid).

Notice this is O(1) work. Draw a picture or two to convince yourself this works.

Note: A simpler approach to answering queries in O(1) time would be to pre-compute the answer to every possible query. But that would take O(x2y2) space and pre-processing time.

Version 4: Smarter and Parallel

As in version 2, the initial corner finding should be done in parallel. As in version 3, you should create the grid that allows O(1) queries. The first step of building the grid should be done in parallel using goroutines. The second step should remain sequential; just use the code you wrote in version 3. Parallelizing it is part of Version 6.

To parallelize the first grid-building step, you will need each parallel subproblem to return a grid. To combine the results from two subproblems, you will need to add the contents of one grid to the other. The grids may be small enough that doing this sequentially is okay, but for larger grids you will want to parallelize this as well. (To test that this works correctly, you may need to set a sequential-cutoff lower than your final setting.)

Version 5: Smarter and Lock-Based

Version 4 may suffer from doing a lot of grid-copying in the first grid-building step. An alternative is to have just one shared grid that different goroutines add to as they process different census-block-groups. But to avoid losing any of the data, that means grid elements need to be

protected by locks. To allow simultaneous updates to distinct grid elements, each element should have a different lock. Use sync.Mutex for this task.

Note you do not need to re-implement the code for finding corners of the country. Use the code from versions 2 and 4. You also do not need to re-implement the second grid-building step. You are just re-implementing the first grid-building step using goroutines, a shared data structure, and locks.

Version 6:
In version 4, the second step of grid-building is still entirely sequential, running in time O(x*y). We can use parallel prefix computations to improve this -- the most straightforward approach involves two different parallel-prefix computations where the second uses the result of the first. The first parallel-prefix works for each row and the second works for each column of the result of the first parallel-prefix. Implement this so that the span of for this grid-building step is O(log x + log y).

Provided Code
The provided code will take care of parsing the input file (sequentially), performing the Mercator Projection, and putting the data you need in arrays.

Main and Command-Line Arguments
Your main method should be in a file called PopulationQuery.go and it should take at least 4 command-line arguments in this order:

- The file containing the input data
- x, the number of columns in the grid for queries
- y, the number of rows in the grid for queries
- One of -v1, -v2, -v3, -v4, -v5, -v6 corresponding to which version of your implementation to use