Populations

Due April 7, 2013

Assignment 9
CSC 062: Spring, 2013

US Census Data

The availability of electronic data is revolutionizing how governments, businesses, and organizations make decisions. But the idea of collecting demographic data is not new. For example, the United States Constitution has required since 1789 that a census be performed every 10 years. In this project, you will process some data from the 2000 census in order to answer efficiently 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 210,000 data points, so any desktop computer has plenty of memory. On the other hand, this size makes using parallelism less compelling (but nonetheless required and educational).

You will implement the desired functionality in several ways that vary in their simplicity and efficiency. Some of the ways will require fork-join parallelism and, in particular, Java's ForkJoin Framework. Others are entirely sequential.

The final portion of this project involves comparing execution times for different approaches and parameter settings. You will want to write scripts or code to collect timing data for you, and you will want to use a machine that has at least 4 processors. The Mac's in the CS department lab in Edmunds 227 typically have 8 cores. In addition, you will need to log in and run your code remotely on `project2.cs.pomona.edu`, which is a 48 core linux machine available in the department.

This project is an experiment where much of the coding details and experimentation are left to you, though we will describe the algorithms you must use. Will parallelism help or hurt? Does it matter given that most of your code runs only in a preprocessing step? The answers may or may not surprise you, but you should learn about parallelism along the way.

Learning Objectives

For this project, you will:

- Gain experience designing software that processes data in several stages
- Write divide-and-conquer parallel algorithms using fork-join parallelism
- Experience the difficulty of evaluating parallel programs experimentally
- Write a report to present the interesting parameters of your implementation and how you evaluated them

The Assignment

You are encouraged to work in pairs on this assignment. If you do choose to work with a partner, you will submit one project for both students. You will also be required to submit a description of how you split the work between you.

Overview of what your program will do

The file `blkgrp_pop_centroid_withname.txt` (distributed with the project files) was published by the U.S. Census Bureau at

`http://www.census.gov/geo/www/cenpop/blkgrp/bg_cenpop.html`

(It is in the .zip file under the "All 50 States" link at the bottom.) The data divides the U.S. into 211,267 geographic areas called "census-block-groups" and reports for each such group the population in 2000 and

the latitude and longitude of the group. It actually reports the average latitude and longitude of the people in the group, but that will not concern us: just assume everyone in the group lived 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 and 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. popluation.
- 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 coordinates for each census group. That is because the Earth is spherical but our grid is a rectangle. 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. You can manually disable this projection while testing by changing the code if you find it helpful to do so.

We can next imagine answering queries related to areas inside the U.S.:

- For some smaller rectangle inside the U.S. rectangle, what is the 2000 census population total?
- For some smaller rectangle inside the U.S. rectangle, what percentage of the total 2000 census U.S. population is in it?

Such questions can reveal that population density varies dramatically in different regions, which explains, for example, how a presidential candidate can win despite losing the states that account for most of the geographic area of the country. By supporting only rectangles as queries, we can answer queries more quickly. Other shapes could be approximated using multiple small rectangles.

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. For testing and timing purposes, you may also wish to provide an alternative where queries are read from a second file.

More details on how your program should work

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

1. The file containing the input data
2. Two integers x and y describing the size of a grid (a two-dimensional array, which in Java means an array of arrays) that is used to express population queries

See the end of this write-up for details on using command line arguments in Eclipse.

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.) Thus, 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 in the following order:

1. The westernmost column that is part of the rectangle; error if this is less than 1 or greater than x.
2. The southernmost row that is part of the rectangle; error if this is less than 1 or greater than y.

3. The easternmost column that is part of the rectangle; error if this is less than the westernmost column or greater than x.

4. The northernmost row that is part of the rectangle; error if this is less than the southernmost column or greater than y.

You program should print a single one-line prompt asking for these four numbers and then read them in. 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 (rounded to two decimal digits, e.g., 37.22U.S. population in the queried rectangle

You should then repeat the prompt for another query.

To implement your program, you will need to determine in which grid position each census-block-group lies. That will require first knowing the four corners of the U.S. rectangle, which can be computed by finding 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.

Four Different Implementations

You will implement up to 4 versions of your program (including the extra credit, if you choose to do it). There are signficant opportunites to share code among the different versions and you should seize these opportunities.

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 most reusable approach for each census-block-group is probably to first compute what grid position it is in and then test 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 the ForkJoin Framework effectively. The work will remain $O(n)$, but the span should be $O(\log n)$. Finding the corners should require only one data traversal, and each query should require only one additional data traversal.

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:

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 $O(n)$ traversal, where n is the number of census-block-groups.

Now modify the grid so that instead of each grid element holding the total for that grid position, it instead holds the total for that position and all positions that are farther north and/or west. In other words, the grid element g stores the total population of everyone in the rectangle whose upper-left corner is the northwest corner of the country and whose lower-right corner is g. This can be done in time $O(xy)$ 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 (with x=y=4):

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 Java 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 $O(xy)$ time 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:

1. Take the value in the bottom-right corner of the query rectangle.
2. Subtract the value just above the top-right corner of the query rectangle (or 0 if that is outside the grid).
3. Subtract the value just left of the bottom-left corner of the query rectangle (or 0 if that is outside the grid).
4. 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 precompute the answer to every possible query ahead of time. But that would take $O(x^2y^2)$ space and preprocessing time.

Version 4 (Extra Credit): 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, building the grid, should be done in parallel using the ForkJoin

Framework. The second step, modifying the grid to sum from the northwest, should remain sequential; just use the code you wrote in version 3.

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. If the grids were always small, doing this sequentially would be okay, but for larger grids you will want to parallelize this as well using another ForkJoin computation. So, in order to merge two results during the parallel grid-building, you will need to add the grids in parallel. (To test that this works correctly, you may need to set a sequential cutoff lower than your final setting.)

Warning: do not attempt to use a single shared grid into which all threads add their results. The result of having two threads writing the same grid cell at the same time is unpredictable!

Note that your ForkJoin tasks will need several values that are the same for all tasks: the input array, the grid size, and the overall corners. Rather than passing many unchanging arguments in every constructor call, it is cleaner and probably faster to pass an object that has fields for all these unchanging values.

Code Provided To You

The code provided to you will take care of parsing the input file (sequentially), performing the Mercator Projection, and putting the data you need in a large array. The provided code uses `float` instead of `double` since the former is precise enough for the purpose of representing latitude and longitude and takes only half the space.

You should avoid timing the parsing since it is slow but not interesting. The rest is up to you. Make good design decisions.

Main Class and Command-Line Arguments

Your main method should be in a class called PopulationQuery and it should take at least 4 command-line arguments in this order:

1. The file containing the input data
2. x, the number of columns in the grid for queries
3. y, the number of rows in the grid for queries
4. One of -v1, -v2, -v3, or -v4 corresponding to which version of your implementation to use

You are welcome to add additional command-line arguments after these four for your own experimentation, testing, and timing purposes, but your command line parameters should be consistent with those specified above for our testing, and a cleaner approach would be to use a different main method in another class.

Help with Command Line Arguments in Eclipse

In Java, command line arguments are passed to the program's `main` method using the `args` array of type `String`. For example, assume you run a program from the command line as follows:

```
> java PopulationQuery filename 10 20 -v4
```

Then in your `main` method, the `args` array will have length of 4, where `args[0]` will hold the value "filename", `args[1]` will hold the value "10", `args[2]` will hold the value "20", and `args[3]` will hold the value "-v4". Note that in all cases, the values are stored as `String`s. Thus, to use the value in `args[1]` as an int, it will need to be converted. For example, by calling `Integer.parseInt(args[1]);`.

In order to pass command line arguments using Eclipse, you need to go to the `Run` menu at the top of the screen. Select `Run Configurations...` from the menu. When the `Run Configuration` window pops up, select the appropriate configuration from the left hand side of the screen. Then click on the "Arguments" tab, and enter your command line arguments in the text box labeled "Program arguments:" separating each argument by a space.

Experimentation

The write-up requires you to measure the performance (running time) of various implementations with different parameter settings. To report interesting results properly, you should use a machine with at least four processors and report relevant machine characteristics.

You will also need to report interesting numbers more relevant to long-running programs. In particular you need to:

- Not time parsing the input data into a file
- Allow the Java Virtual Machine and the ForkJoin Framework to "warm up" before taking measurements. The easiest way to do this is to put the code to be timed in a loop and not count the first few loop iterations, which are probably slower. That is, to test the running time of some variant V, execute something like

  for (i = 0; i < 15; i++) V

  and average the time of the last 10 iterations. The first few iterations may run slower, until Java decides to invest the effort to optimize V. That is why you ignore the first iterations. Even once this optimization is done, each iteration of V may take a somewhat different amount of time, which is why it is more accurate to take an average of several runs of V. While this is wasted work for your program, (a) you should do this only for timing experiements and (b) this may give a better sense of how your program would behave if run many times, run for a long time, or run on more data.
- Write extra code to perform the experiments, but this code should not interfere with your working program. You do not want to sit entering queries manually in order to collect timing data.

For guidelines on what experiments to run, see the Write-Up Questions. Note you may not have the time or resources to experiment with every combination of every parameter; you will need to choose wisely to reach appropriate conclusions in an effective way.

Write-Up Questions

Turn in a report answering the following questions. Note there is a fair amount of data collection for comparing timing, so do not wait until the last minute. Prepare an actual report, preferably a PDF file, but we will also take other common formats such as Microsoft Word.

1. If you worked with a partner, how did you split up the work? Describe who did what work.

2. How did you test your program? What parts did you test in isolation and how? What smaller inputs did you create so that you could check your answers? What boundary cases did you consider? Be sure to describe the system you used for testing. What type of machine did you use? How many cores were used in the test? (On a mac, you can open the "Activity Monitor" program. Then go to the "Window" menu item, and select "CPU Usage." This should give you a floating window that shows the usage of each of the cores.

3. Compare the performance of version 1 to version 2 and version 3 to version 4 (if implemented). Did parallelism help?

4. For finding the corners of the United States and for the first grid-building step, you implemented parallel algorithms using Java's ForkJoin Framework. The code should have a sequential cut-off that can be varied. Perform experiments to determine the optimal value of this sequential cut-off. Graph the results and reach appropriate conclusions. Note that if the sequential cut-off is high enough to eliminate all parallelism, then you should see performance close to the sequential algorithms, but evaluate this claim empirically.

5. Compare the performance of version 1 to version 3 and version 2 to version 4 (if implemented) as the number of queries changes. That is, how many queries are necessary before the preprocessing is worth it? Produce and interpret an appropriate graph or graphs to reach your conclusion. Note you should time the actual code answering the query, not including the time for a (very slow) human to enter the query.

6. What Extra Credit projects did you implement? What was interesting or difficult about them? Describe how you implemented them.

Acknowledgement

This project was created in Spring 2010 by Dan Grossman. A full write-up of the assignment appears here (including some GUI enhancements by Brent Sandona that are not required for this assignment, but may be interesting):

`http://www.cs.washington.edu/education/courses/cse332/11au/homework/asst6.html`

We have made slight modifications to the project for this class, but have tried to duplicate all of the relevant information. You may find it useful to review the original.