Last Updated: Feb 17, 2013

CSE 332 Winter 2013: Project 3 - Where are the People?

Outline

Due Dates and Turn-In

- Find a partner: As soon as possible, you must notify us before Wednesday Feb 27 if you plan on working with a partner
- Phase A: Tuesday March 5, 11PM (10% of project 3 grade)
- Phase B: Tuesday March 12, 11PM (65% of project 3 grade)
- Write-up: Thursday March 14, 11PM (25% of project grade)

We do not recommend doing all the work necessary for the write-up in the last two days. You may need to modify your code by adding timing calls, etc., after March 12. That is fine, but your code will be graded on the March 12 submission.

Use the course dropbox to submit all parts of your assignment electronically.

Working with a Partner

Instructions regarding partners are the same as in Project 2.

You are strongly encouraged, but not required, to work with a partner of your own choosing for this project. You may work with somebody you already know or use the course message board to help find a partner. No more than two students total may form a team. You may divide the work however you wish, under three conditions:

- You document each team member's effort in your write-up.
- You work together and make sure you *both* understand the answers to all write-up questions.
- You understand how your partner's code is structured and how it works.

Other logistics:

- Turn-in only one submission per team (i.e., only one set of Phase A code, only one set of Phase B code, only only one Writeup). It would help the course staff if the same partner submits all of these via their Catalyst account, but this is not required.
- Both names should appear in all files, although it is fine to attribute one partner as a primary author on a particular piece of code.

Test all of your code together to be sure it properly integrates. Start this early, and do not attempt to merge your code on the due date. You will very likely experience problems with integration. You may wish to set up an SVN repository to make collaboration easier. If so, contact the course staff regarding project space.

Team members will receive the same project grade unless there is an *extreme circumstance* and you notify us *in advance of the deadline*.

If you plan to work with a partner, one partner MUST send Daniel an email using this link. In your email, provide the following information for BOTH partners: full name, UWNetID, CSE Email address.

Introduction

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 2010 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 220,000 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 last (not necessarily best) approach uses explicit threads, a shared data structure, and lock-based synchronization.

A final portion of this project involves comparing execution times for different approaches and parameter settings. You will want to write scripts to collect timing data for you, and you will want to use a machine that has at least 4 processors.

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 pre-processing 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
- Investigate a different implementation approach that requires locking for synchronization
- Experience the difficulty of evaluating parallel programs experimentally
- Write a report to present the interesting parameters of your implementation and how you evaluated them

Provided Code

You need several files provided in this single zip file:

project3files.zip

- You do *not* need to look at any of the files that implement the optional graphical interface provided to you: `InteractionPane.java`, `MapPane.java`, `USMaps.java`, `USMap.jpg`, `contUSmap.jpg`
- You *do* need to understand these files: `CensusData.java`, `CensusGroup.java`, `Rectangle.java`, `PopulationQuery.java`, `CenPop2010.txt`
- To use the graphical interface, you need to understand the simple `Pair.java` file.

You will also find the Introduction to the ForkJoin Framework (JSR 166) useful.

The Assignment

Overview of what your program will do

The file `CenPop2010.txt` (distributed with the project files) 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. 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 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. If you find it helpful to do so, you can change the code to disable this projection during your testing.

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

1. For some rectangle inside the U.S. rectangle, what is the 2010 census population total?
2. For some rectangle inside the U.S. rectangle, what percentage of the total 2010 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 country's geographic area. By supporting only rectangles as queries, we can answer queries more quickly. A different shape can be approximated using multiple rectangles, but this is "Above & Beyond."

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. We also provide you a graphical interface that makes asking queries more fun.

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, which in Java means an array of arrays) 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 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 of the U.S. population in the queried rectangle, rounded to two decimal digits, e.g., 37.22%

You should then repeat the prompt for another query.

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.

Five Different Implementations

You will implement 5 versions of your program. There are signficant opportunites to share code among the different versions and you should seize these opportunities. So dividing the work with a partner by splitting up the versions may not work well.

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 the ForkJoin Framework effectively. 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.

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 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(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 $O(x^2y^2)$ space and pre-processing time, and is not acceptable for version 3 of your program.

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 the ForkJoin Framework. The *second* step should remain sequential; just use the code you wrote in version 3. Parallelizing it is part of the Above & Beyond.

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 are required to ~~will want to~~ parallelize this as well using another ForkJoin computation. (To test that this works correctly, you may need to set a sequential-cutoff lower than your final setting.)

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.

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 threads 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.

In version 5, you will implement this strategy. You should *not* use the ForkJoin Framework; it is not designed to allow synchronization operations inside of it other than `join`. Instead you will need to take the "old-fashioned" approach of using explicit threads. It is okay to set the number of threads to be a static constant, such as 4.

How you manage locks is up to you. You could have the grid store objects and lock those, or you could have a separate grid of just locks. Note that after the first grid-building step, you will not need to acquire locks anymore (use `join` to make sure the grid-building threads are done!).

Note you do not need to re-implement the code for finding corners of the country. Use the ForkJoin Framework 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 Java threads, a shared data structure, and locks.

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 a large array. The provided code uses `float` instead of `double` since the former is precise enough for the purpose of representing latitude/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`, `-v4`, `-v5` 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 a cleaner approach is likely to use a different `main` method in another class.

Regardless of any extra command line parameters you may add, you should ensure that your program will work when called from the main method in `PopulationQuery`. To facilitate the grading process, your program should **exactly** match the format shown below (note that `>>` indicates user input, but should not appear in your output):

```
>>java PopulationQuery CenPop2010.txt 100 500 -v1
Please give west, south, east, north coordinates of your query rectangle:
>>1 1 100 500
population of rectangle: 312471327
percent of total population: 100.00
Please give west, south, east, north coordinates of your query rectangle:
>>1 1 50 500
population of rectangle: 27820072
percent of total population: 8.90
Please give west, south, east, north coordinates of your query rectangle:
>>exit
```

When your program sees any input that is not 4 integers on a line, exit the program without printing any additional output.

Also see below for how to write methods that the graphical interface can call; it does not call your `main` method. You can use the graphical interface with some of your versions even if others are not yet implemented.

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 initial processing in a loop and not count the first few loop iterations which are probably slower. 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.

Submission

Submit all your new files, including any additional Java files you created for testing, and any provided files you modified. Make sure your code is properly documented, etc. We are not specifying how to test your code, but we still want you to test your programs and show us how you did so.

Phase A, due Tuesday March 5

> Submit all of your files for Version 1 and Version 2. This should include any additional files you created for testing. Style must be reasonable in this submission, but it will be primarily graded for correctness.

Phase B, due Tuesday March 12

> **Submit *all* of your files for all five versions** (including files you submitted with Phase A). **This should include any additional files you created for testing.** It will be graded for correctness and style. Submit and "Above & Beyond" code in an `extracredit.zip` file.

Writeup, due Thursday March 14

> You may submit a `txt` file with separate image files for your charts. You may also insert your charts inline in an `html` or `pdf` file. Formatting is not a component of your grade, as long as the report is clear and readable.
>
> **You must submit all of your code using for timing** in the preparation of your report. This code will **not** be graded and is only to help us understand what you did if there are any questions. The need to time your code should not impact the quality of your design.

The Graphical Interface

The provided graphical user interface (GUI) for the program is fun (we hope), easy to use, and useful for checking

your program against some geographical intuition (e.g., nobody lives in the ocean and many people live in Southern California).

The GUI presents a map of the U.S. as a background image with a grid overlaid on it. You can select consecutive grid squares to highlight arbitrary rectangles over the map. When you select run, the GUI will invoke your solution code with the selected rectangle and display the result.

To run the GUI, run the main method of the class `USMaps`. (If you are using Java 6 instead of Java 7, you still need the VM argument `-Xbootclasspath/p:jsr166.jar`.)

In the GUI, you can "zoom in" to the continental U.S. When zoomed, keep in mind two things:

- Zooming means the entire grid is not shown. For example, if the grid has 12 rows and 23 columns, zooming will show 6 rows (with most of the bottom one not shown) and 13 columns. So even selecting all the *visible* grid rectangles will not select all the *actual* grid rectangles.
- If you select partly-viewable grid rectangles on the edge, this selects the entire grid rectangle, *including* the population for the census-block-groups in the not-visible portion of the grid rectangle. For small grid sizes, this can include portions of Puerto Rico and for very small (silly) grid sizes, it can include Alaska or Hawaii.

Naturally, the GUI needs to call your code and it can do so only if you implement an API that the GUI expects. To use the GUI, you must write two methods in the class `PopulationQuery` with the following signatures:

- `public static void preprocess(String filename, int x, int y, int versionNum);`
- `public static Pair<Integer, Float> singleInteraction(int w, int s, int e, int n);`

The arguments to the `preprocess` method are the same arguments that should be passed via the command line to the main method in `PopulationQuery`, only parsed into their datatypes and not as Strings. This method should read the file and prepare any data structures necessary for the given version of the program. The arguments to the `singleInteraction` method are the arguments that are passed to the program when it prompts for query input. This method should determine the population size and the population percentage of the U.S. given the parameters, just as your program should when given integers at the prompt.

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.

1. Who is in your group? (Only one writeup needs to be submitted per group.)
2. What assistance did you receive on this project? Include anyone or anything *except* your partner, the course staff, and the course materials / textbook.
3. How long did the project take? Which parts were most difficult? How could the project be better?
4. What "Above & Beyond" projects did you implement? What was interesting or difficult about them? Describe how you implemented them.
5. 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?
6. 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.
7. Compare the performance of version 4 to version 5 as the size of the grid changes. Intuitively, which version is better for small grids and which version for large grids? Does the experimental data validate this hypothesis? Produce and interpret an appropriate graph or graphs to reach your conclusion.
8. Compare the performance of version 1 to version 3 and version 2 to version 4 as the number of queries changes. That is, how many queries are necessary before the pre-processing 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.
9. If you worked with a partner:

a. Describe the process you used for developing and testing your code. If you divided it into pieces, describe that. If you worked on everything together, describe the actual process used. For example, discuss how long you talked about what, in what order you wrote and tested the code, and how long that required.
b. Describe each group member's contributions/responsibilities in the project.
c. Describe at least one good thing and one bad thing about the process of working together.

Above & Beyond

You may do any or all of the following; pick ones you find interesting.

1. *Parallel prefix:* 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. Implement this so that the span of for this grid-building step is O(log x + log y). Run experiments to determine the effect of this change on running time.
2. *Earth curvature:* Our program uses the Mercator Projection, which badly distorts locations at high latitude. Either use a better projection (read about map projections on your own) or properly treat the Earth as a sphere. Adjust how queries are handled appropriately and document your approach. *Do all this in separate files so that we can still grade your rectangular version.*
3. *Polygonal queries:* Write a different `main` method that supports queries that are arbitrary polygons instead of rectangles. For a polygon, the user can enter any set of grid positions and the polygon should be the shape that connects these points in order (connecting the last back to the first). Reject a query in which any lines cross each other. Then answer the query by transforming it into as few rectangular queries as possible.
4. *Other:* We welcome additional suggestions for Above & Beyond items. Contact the course staff with your idea.

Acknowledgments

This project was created in Spring 2010 by Dan Grossman. Brent Sandona created the GUI and Jacob Sanders added the support for zooming in on the continental U.S. Dan got several good ideas and pieces of feedback from other faculty members (Alan Borning, James Fogarty, Hal Perkins, Larry Snyder, maybe others) while he was figuring out the project. Martin Tompa improved the write-up in Fall 2010.