

P3: Where are the People?**Checkpoint 1:** Due Tue, November 17**Checkpoint 2:** Due Tue, December 1**P3 Due Date:** Due Mon, December 7

The purpose of this project is to get familiar with writing sequential code and then using parallelism to potentially speed up that code. You will be using the ForkJoin framework and Java threads to achieve parallelism.

Overview

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 2,100,333 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.

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

- For some rectangle inside the U.S. rectangle, what is the 2010 census population total?
- 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 than if more complex shapes were used.

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 have provided the code to take care of parsing the input file (sequentially), performing the Mercator Projection, and putting the data into a large array as well as printing the query responses. You will write the algorithms that support all of the code we've written.

IMPORTANT NOTICE: For this project, we will not be using Gitlab for submission of the project, only to distribute and store the project. You will turn in your code to Gradescope along with the write-up. There are also no Gitlab pipelines or static analysis for this project.

Project Restrictions

- You *must* work in a group of two unless you successfully petition to work by yourself.
- You may not use **any** of the built-in Java data structures or related methods like `Arrays.copyOf()`. One of the main learning outcomes is to write everything yourself.
- You may use the `math` package.
- You may not edit any file in the `cse332.*` packages.
- The *design and architecture* of your code are important, but will not be directly graded.
- The Write-Up is a *substantial* part of your grade; do **not** leave it to the last minute.
- **DO NOT MIX** any of your experiment or above and beyond files with the normal code. Before changing your code for experiments or above and beyond, copy the relevant files into the corresponding package (e.g., `aboveandbeyond`, `experiments`). If your code does not compile because you did not follow these instructions, you will receive a 0 for all automated tests.
- Make sure to not duplicate fields that are in super-classes (e.g., `size`). This will lead to unexpected behavior and failures of tests.

Provided Code

IMPORTANT NOTICE: Do **NOT** modify anything in the provided code as it will break the tests. If you do modify something by accident and cannot undo it, let staff know or go back to the Git history to recover it.

- `cse332.interfaces`
 - `QueryResponder.java`: Defines a class that responds to population queries. It also stores the total population and has a getter method for it.
- `cse332.types`
 - `CensusData.java`: A data structure that holds multiple `CensusGroups`. It is essentially an array.
 - `CensusGroup.java`: A single census data point, storing latitude, longitude, and population for that location.
 - `CornerFindingResult.java`: A simple pair object that holds a `MapCorners` and an `int`. It is intended to be used as a return type for `CornerFindingTask.java`.
 - `MapCorners.java`: `MapCorners` represents a rectangular space defined by latitude and longitude. It has a useful `encompass` method to combine two `MapCorners` objects.
- `main`
 - `PopulationQuery.java`: The main driver program that gets user queries. It also parses the census data

You will implement `SimpleSequential`, `SimpleParallel`, `ComplexSequential`, `ComplexParallel`, and `ComplexLockBased` (in `queryresponders`). You will also need to implement `CornerFindingTask`, `GetPopulationTask`, `PopulateGridTask`, `MergeGridTask` and `PopulateLockedGridTask` (in `paralleltasks`) which are used in the parallel query responders.

How does Population Query work?

The main way you will be interacting with your program is through `PopulationQuery.java`. You will be passing the following 4 arguments to the main method of `PopulationQuery.java`:

1. The file containing the input data

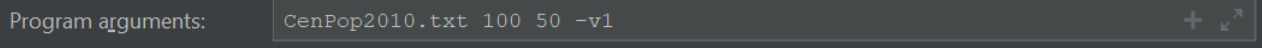
For the 2010 US Census data, this will be `CenPop2010.txt`.

2. Two integers `numColumns` and `numRows` describing the size of a grid that is used to execute user queries. This grid will be an abstract idea that partitions the U.S. map, and you will need to determine within which *grid cell* each census-block-group lies as you implement your query responders. In the unlikely case that a census-block-group falls exactly on the border of more than one *grid cell*, tie-break by assigning it to the North and/or East.

Let's say that `numColumns = 100` and `numRows = 50`. That would mean we want to think of the 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 $100 \cdot 50 = 5,000$ little rectangles in it (a *grid cell*). Providing larger values for `numColumns` and `numRows` will let us answer queries more precisely but will require more time and/or space.

3. One of `-v1`, `-v2`, `-v3`, `-v4`, `-v5` corresponding to which version of your implementation to use.

For example, the following arguments would read the input file `CenPop2010.txt`, allow the user to specify queries using a grid containing 100 columns and 50 rows, and run version 1 of your implementation (`SimpleSequential`):



```
Program arguments: CenPop2010.txt 100 50 -v1
```

To insert command line arguments to IntelliJ, run `PopulationQuery.java` first (it should give an error about not having enough arguments). Next, go to `Run | Edit Configurations`. On the left side of the window that pops up, click on `PopulationQuery` then insert as shown above into `Program arguments`.

After reading these arguments, `PopulationQuery.java` will prompt the user in the terminal, asking what *query rectangle* they would like `PopulationQuery.java` to provide population data for:

```
>> Please give west, south, east, north coordinates of your query rectangle:  
>> west south east north
```

A *query rectangle* is simply four integers that describes a rectangle within the US map that we want to grab population data from. These integers must be valid rows and columns:

1. **west**, the western-most grid column that is part of the query rectangle; error if this is less than 1 or greater than `numColumns`.
2. **south**, the southern-most grid row that is part of the query rectangle; error if this is less than 1 or greater than `numRows`.
3. **east**, the eastern-most grid column that is part of the query rectangle; error if this is less than the Western-most column (equal is okay) or greater than `numColumns`.
4. **north**, the northern-most grid row that is part of the query rectangle; error if this is less than the Southern-most column (equal is okay) or greater than `numRows`.

The program will then use the requested version of your implementation to calculate the query and output 3 numbers:

1. The total population in the U.S.
2. The total population in the query rectangle
3. The percentage of the U.S. population in the query rectangle, rounded to two d.p. (e.g. 8.90%)

If numColumns is 100 and numRows is 50, the following are examples of querying the entire grid (1 1 100 50) and then half the grid (1 1 50 50):

```
>> Please give west, south, east, north coordinates of your query rectangle:
>> 1 1 100 50
>> population of the U.S.: 312471327
>> population of rectangle: 312471327
>> percent of total population: 100.00
>> Please give west, south, east, north coordinates of your query rectangle:
>> 1 1 50 50
>> population of the U.S.: 312471327
>> population of rectangle: 27820072
>> percent of total population: 8.90
>> Please give west, south, east, north coordinates of your query rectangle:
>> this can be anything - just press enter and the program will exit
>>
>> Process finished with exit code 0
```

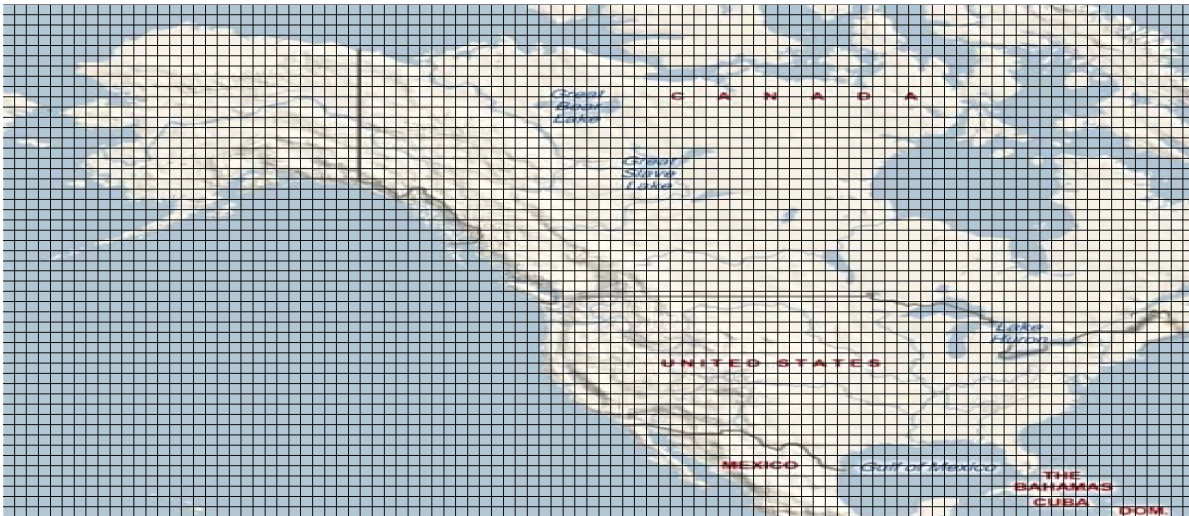
Note: PopulationQuery.java will safely exit if it sees **anything** that isn't 4 integers.

By this point, this is the image that you should visually have in mind for the grid:

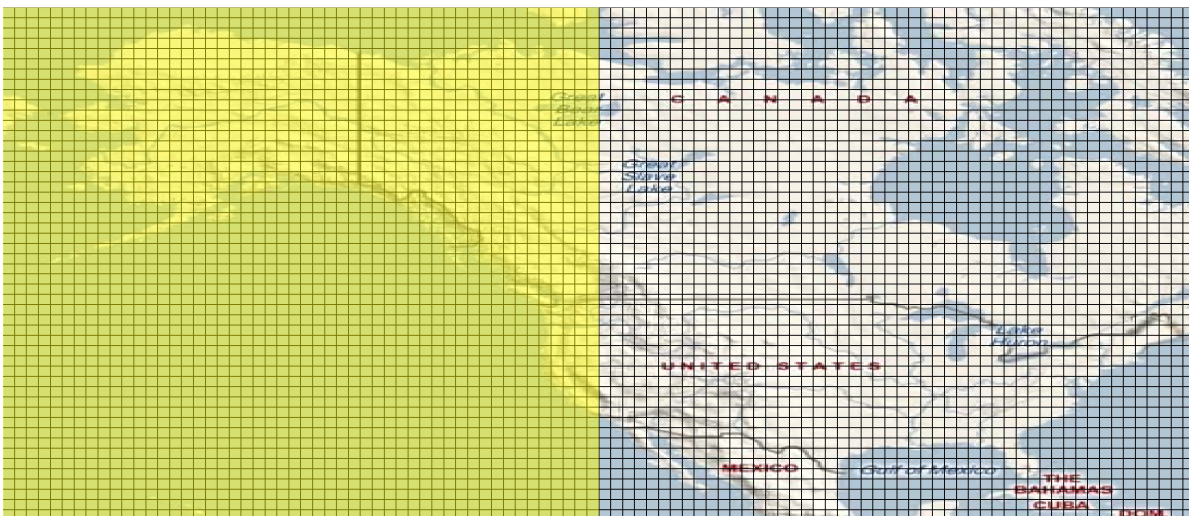
- (a) We start with the US map warped using the Mercator Projection (notice how the northern parts of the map appear larger than the southern parts)



- (b) We have the numColumns and numRows so we can form the grid in mind (in this case, we use the same example of 100 numColumns and 50 numRows)



- (c) Finally, we pass in the 4 integers that represent the query rectangle (in this case, we arbitrarily choose 1 1 50 50 - this is half the grid)



From there, you will calculate what the population of the query rectangle (highlighted yellow) is.

Project Checkpoints

This project will have **two** checkpoints (and a final due date). A *checkpoint* is a check-in on a certain date to make sure you are making reasonable progress on the project. For each checkpoint, you (and your partner) will turn in a Gradescope survey individually.

As long as you turn in the checkpoint survey, the checkpoint will not affect your grade in any way.

Checkpoint 1: (1), (2)

Tue, November 17

Checkpoint 2: (3), (4)

Tue, December 1

P3 Due Date: (5), (6)

Mon, December 7

Part 1: Simple Brute Force Algorithms

(1) Simple & Sequential

To make `PopulationQuery.java` work, you will work on 5 different versions of `QueryResponders`. This will be the first implementation (version 1) of a `QueryResponder`. Take a look at `QueryResponder.java`. In every version of `QueryResponder`, you will be implementing both the constructor (the **preprocessing** step) and the `getPopulation()` method (the **calculating** step).

The constructor for each of these `QueryResponders` will take in a `CensusData` object, and two integers `numColumns` and `numRows` (due to Java restrictions, we cannot enforce a constructor and its parameters but take a look at `SimpleSequential.java` to see the parameters). The `CensusData` object will hold all the data parsed from the 2010 U.S. Census. Take a look at `CensusData.java` to see how the data is organized. We suggest printing out the different fields to visually see what the data looks like. The two integers `numColumns` and `numRows` correspond to the same two integers passed into `PopulationQuery.java`. For each of these versions, you will create a constructor that ensures the `totalPopulation` field is correctly updated to the total U.S. population and also does the preprocessing (if any) required by this version. This preprocessing step will be different in every version.

The `getPopulation()` method will allow `PopulationQuery.java` to get the population at a specified query rectangle, which will be the 4 int parameters `west`, `south`, `east`, and `north`. These correspond to the 4 integers passed into `PopulationQuery.java`. Using these 4 values, you will calculate and return what the population of the query rectangle is. Again, this calculating step will be different in every version.

In `SimpleSequential.java`, you will be implementing version 1 of a `QueryResponder`.

In the constructor, 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. You are free to do this in any way you think is best but we suggest using the `MapCorners.java` class to help you process the `CensusData` more easily.

In the `getPopulation()` method, do another sequential $O(n)$ traversal to answer the query by determining if each census-block-group is in the query rectangle. The simplest and most reusable approach for each census-block-group is probably to first compute which grid cell it belongs to and then see if that grid cell is in the query rectangle. To do this you may need to store some additional fields and calculate them in the constructor. **Important note: make sure you are using double's instead of float's to calculate or store latitude and longitude (except in `CensusData`) as there may be rounding issues when using float's.**

(2) Simple & 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 which will happen in parallel, and each query should require only one additional data traversal, also in parallel.

For this part, you will need to implement `SimpleParallel.java` as well two ForkJoin tasks found in the `paralleltasks` folder of `src`, `CornerFindingTask.java` and `GetPopulationTask.java`.

Part 2: Preprocessed Grid Algorithms

(3) Complex & 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 `numRows · numColumns` (use an array of arrays) where each element is an `int` that will hold the total population for that *grid cell*. Recall `numColumns` and `numRows` are the command-line arguments for the grid size. Compute the grid using a single traversal over the input data.
- Now modify the grid so that instead of each grid element holding the total for that *grid cell*, it instead holds the total for all *grid cell*'s to the west and to the south of the current *grid cell*. In other words, in the modified grid, the *grid cell* (x, y) stores the total population of the rectangle whose lower-left corner is the South-West corner of the country $(1, 1)$ and the upper-right corner is (x, y) . This can be done in time $O(\text{numColumns} \cdot \text{numRows})$ but you need to be careful about the order you process the elements.

For example, if `numColumns` and `numRows` are both 4, suppose after step 1 we have this grid containing the total population for each *grid cell*:

$$\begin{bmatrix} 9 & 1 & 1 & 1 \\ 2 & 2 & 0 & 0 \\ 1 & 7 & 4 & 3 \\ 0 & 11 & 1 & 9 \end{bmatrix}$$

Then step 2 would update the grid to be (e.g. 24 is $1 + 7 + 4 + 0 + 11 + 1$):

$$\begin{bmatrix} 12 & 33 & 39 & 52 \\ 3 & 23 & 28 & 40 \\ 1 & 19 & 24 & 36 \\ 0 & 11 & 12 & 21 \end{bmatrix}$$

There is an arithmetic trick to completing the second step in a single pass over the grid. Suppose our *grid cells* 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:

$$\begin{bmatrix} (1, 4) & (2, 4) & (3, 4) & (4, 4) \\ (1, 3) & (2, 3) & (3, 3) & (4, 3) \\ (1, 2) & (2, 2) & (3, 2) & (4, 2) \\ (1, 1) & (2, 1) & (3, 1) & (4, 1) \end{bmatrix}$$

Now, using standard Java array notation, notice that after step 2, for any element not on the left or bottom edge: `grid[i][j]=grid[i][j]+grid[i-1][j]+grid[i][j-1]-grid[i-1][j-1]`. So you can do all

of step 2 in $O(\text{numColumns} \cdot \text{numRows})$ by simply proceeding one grid row at a time bottom to top – or one grid 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 preprocessing step, we can instead get 7 as $40 - 21 - 23 + 11$. In general, the trick is to:

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

Notice this is $O(1)$ work. Drawing a picture will help 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 $O(\text{numColumns}^2 \cdot \text{numRows}^2)$ space and pre-processing time, and is not acceptable for version 3 of your program.

(4) Complex & Parallel

As in version 2, the initial corner finding should be done in parallel. You should use `CornerFindingTask.java` that you wrote in part 2 to accomplish this. 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, where you calculate sums, 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. This is implemented in `PopulateGridTask.java`. To combine the results from two subproblems, you will need to add the contents of one grid to the other. Although for small grids doing this sequentially may be okay, you will need to parallelize this as well using another ForkJoin computation in `MergeGridTask.java`.

Part 3: Locks & The Write-Up

(5) Complex & 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 is modified by different threads as they process different census-block-groups. To avoid losing any of the data, grid elements need to be protected by locks and to allow simultaneous updates to distinct grid elements, each element should have a different lock.

In version 5, you will implement this strategy in `ComplexLockBased.java`. 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 Java 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 (that contain a field for the value at that *grid cell*) and lock those objects, or you could have a separate grid of just locks. You may

modify the fields of the constructor in `PopulateLockedGridTask.java` if you choose to go with the former approach. You may **NOT** use the `synchronized` keyword as a workaround for 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 that 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. You should write your parallel code in `PopulateLockedGridTask.java`

The structure of your parallel thread class will be slightly different from your other `ForkJoin` code. For starters, you will be overriding the `run()` method instead of `compute()`. You may also notice that you need to do a bit of exception handling when joining the threads. In the case that `join()` throws an exception, you should catch it, print the stack trace using `printStackTrace()`, and exit using `System.exit(1)`.

Also notice that there is no sequential cutoff inside `PopulateLockedGridTask.java`. We are now managing the number of threads we have running at a time, as indicated by the `NUM_THREADS` constant inside `ComplexLockBased.java`. Because of this, the `run()` method should process all the census groups between the low and high cutoffs during execution. To ensure the thread requirement is met, make sure you are splitting off `NUM_THREADS - 1` threads and running the last parallel task in the current thread.

(6) Write-Up

A significant portion of your grade will be based on your write-up. You will find the write-up questions here in the [P3 Write-up Template](#). Remember to follow the instructions on the first and second page to receive full credit!

Some of the write-up questions will ask you to design and run some experiments to determine which implementations are faster for various inputs. Answering these questions will require writing additional code to run the experiments, collecting timing information and producing result tables and graphs, together with relatively long answers. Do not wait until the last minute!

Insert tables and graphs into your write-up as appropriate, and be sure to give each one a title and label the axes for the graphs. **IMPORTANT: Place all your timing code into the package `experiment`.** Be careful not to leave any write-up related code in the normal files. To prevent losing points due to the modifications made for the write-up experiments, you should copy all files that need to be modified for the experiments into the package `experiment`, and start working from there. Files in different packages can have the same name, but when editing be sure to check if you are using the correct file! If your code does not compile because you did not follow these instructions, you will receive a 0 for all automated tests.

Above and Beyond

For this project you can find the Above and Beyond parts in the write-up template linked above.

DO NOT MIX any of your above and beyond files with the normal code. Before changing your code for above and beyond, copy the relevant files into the `aboveandbeyond` package. If your code does not compile because you did not follow these instructions, you will receive a 0 for all automated tests.