Ethan Lewis
Zack Rossman
11/12/17

Assignment 9 Report

1) We worked on Zack's computer which is a Lenovo, not a Mac, so Zack did most of the typing because Ethan wasn't used to the interface/keyboard. For each version, we would separately draw out strategies and then come together to put the code together. Zack did most of the J-Unit and main-method testing which Ethan was mostly responsible for this report.

2) We created a J-Unit test to make sure that versions 1 and 2 ran correctly. To make sure that Version 1 summed the population correctly, we gathered a few pieces of data:

· The total US population for the 2000 census, found by iterating through each CensusGroup object in a CensusData object and adding the population to an accumulator

· The US population for each of the 4 quadrants of the US, found by creating a new SumPopulation object and summing the population for each of the queries representing 1 corner of the US.

We then added the population of the 4 quadrants together and asserted that this sum should equal the population found by the accumulator. We also made sure that the query corresponding to the lower right quadrant of the US had the greatest population and that the top right population was 0, since there is no US territory in that area of the grid.

For Version 2, we did the same thing except the population of each quadrant was calculated calling compute(), so that the program would run in parallel.

For Version 3, we created a main method that displayed the population of our 4-quadrant grid in two ways:

· Using the parallel method from Version 2 which displayed the individual populations of each quadrant

· Using Version 3, which displayed the cumulative population of the current quadrant, and all quadrants to the southwest.

· NOTE: our version3 program sums the population starting in the lower left corner (instead of the top left) so that the cumulative population ends up in the top right corner (instead of the lower right). We did this because it was easier to think about starting at position (0,0) in the grid and finishing at position (grid.length, grid.length).

Each quadrant in the version 3 program accurately reflected the sum of quadrants to the southwest.

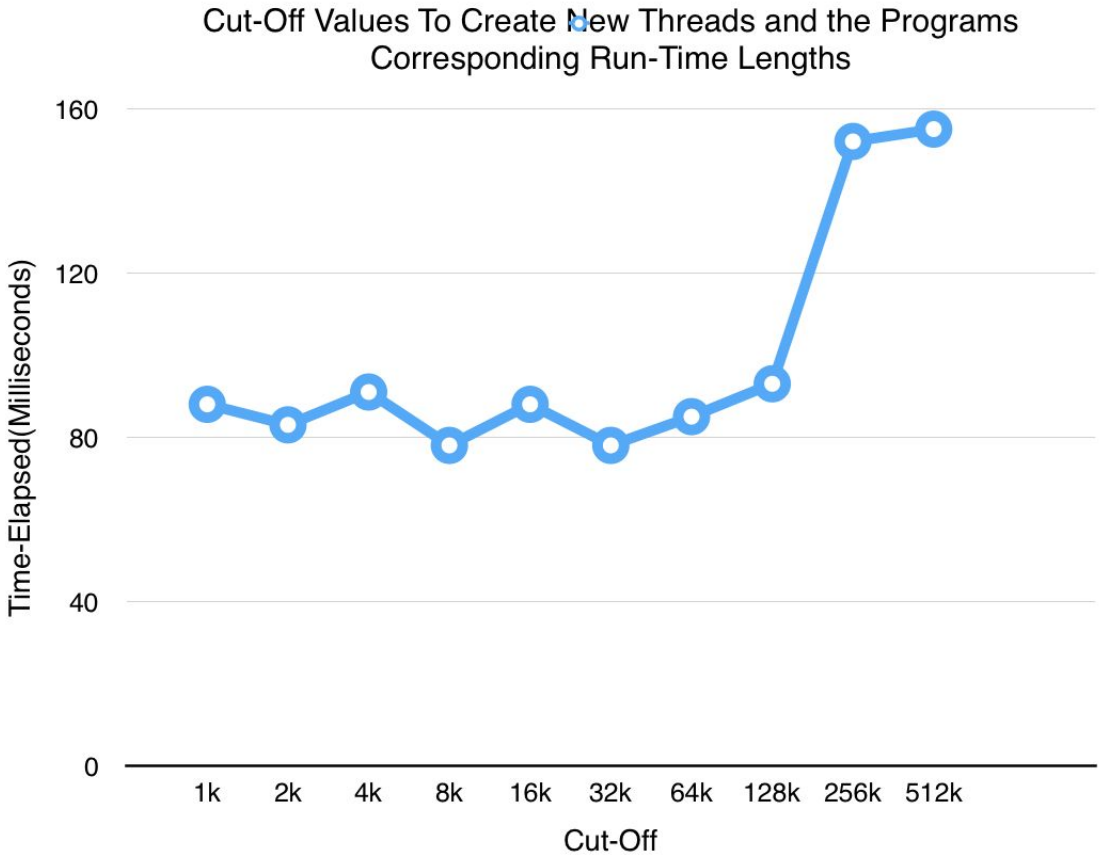
For version 3, we considered boundary cases including when the query rectangle included the leftmost column or the bottommost row in the existing grid. This is important because we don't need to subtract certain rectangles if this is the case. We also tested grabbing data from just one rectangle, from a specific rectangle, and from the entire grid.

We used Zack's computer (Intel Core i7) which has 2 cores and 4 processors.

3) Results from testing:
Version 1: 251 milliseconds
Version 2: 78 milliseconds average after ten warm-up calls
Version 3: 0 milliseconds

Based off the numbers above, version2 with parallelism and a 1000 sequential cutoff was much faster than version1 without parallelism. Note that this was the time to find the specified query not, to construct the grid.

4)



Based off the chart above, we can conclude that the optimal cutoffs are around 8,000 to 32,000 elements. The graph stays relatively even for cutoffs under 130,000 but greatly increases in time when the cutoffs are high or above the number of elements in the array.

5) From the graph below, we can conclude that after 3 queries are added, it will definitely be worth it to pre-process the grid. The time grows exponentially for v1 as the queries double, while v3 stays at a constants around 300 milliseconds.