Sean McClure

Election Simulation Report

COMP 398
Parallel & Distributed
Computing
May 9, 2020

Summary

The algorithm

I designed my election simulation algorithm to use object oriented programming in *C++*, in order to create a clear and structured program. I created two classes, the `State` base class and the `Headquarters` subclass. Fifty `State` objects and one `Headquarters` object are created, where the the latter is the central polling location.

Each state has variables that describe itself, including its name, the number of registered voters, the number of electoral college votes, the postal zone, and its political bias.

The `State` class also has methods, which perform the voting operations for each respective state object. The `vote_count()` function simulates the voting process. It updates the state's vote counts, and returns true in the case of a Red majority, and false for a Blue majority. From there, the `send_delegates()` function returns the status of the Boolean, the number of electoral votes for the state, and the number of Red and Blue votes returned from the simulation.

To determine how each voter within each state object votes two possible methods are used. The first method uses a *political party percentage*, which represents the political bias for each state. Every voter has a probability equal to their respective state's political party percentage to vote Blue. So, for example the voters in a predominantly Red state, such as Texas, are more likely to vote Red than a more Blue state, like Massachusetts. This was done to emulate the polarized political environment of the United States, and the percentages were taken from the results of the 2016 Presidential Election. While this method emulates the political structure of the U.S, it is highly deterministic, meaning the simulation returned the same results each time.

The second method of vote calculation is a more simple coin flip method. Each voter has a 50% chance of voting for either candidate, regardless of what state they are in.

The `Headquarters` class has additional methods used for the aggregation of each individual states data. The `Headquarters` object updates the cost, time, and votes that are calculated from each state. The `Headquarters` object then takes the data from each state and aggregates it, calculating the result of the election, the cost, and the elapsed time.

Rationale

The reason for implementing the simulation in this manner was twofold. The first was that, using object oriented programming naturally emulated the structure of the United States. In the simulation, each object represents a state which is autonomous from the others. U.S states are similar in structure but have their own traits, just as the objects share the same methods, but have different variables/values.

The second reason was that object oriented programming creates an organized structure, as the process is broken up into many small pieces. This makes both programming, and

dissecting the program easier.

One drawback to this method is that, since there is overhead from creating 51 objects, this version is potentially slower than a procedural implementation. However, I believe the benefits of this method, in both its analogous-to-real-life structure, and its code structure outweigh this drawback.

OpenMP was used to parallelize the code, as this method is straightforward to implement, and maintains the format of the original code. Eight OpenMP threads were used, as this produced the most time efficient output. Complete timing data and further analysis is included in the *Performance Report* spreadsheet.

Election Results

The simulation was run with a turnout rate of 0.9, with a central location of Alabama.This location was chosen as it is located in zone 4, which minimizes the mailing time. Statewide, and full results can be found in the *Election Sim Results* spreadsheet.

The result of the simulated election depends on the method of vote calculation. Using the political party percentage method the overall result of the election remains unchanged for each run. The results are stable because the range of variance produced from the random number generation is low, and as most of the percentages are far from the 50% flipping point, it is very unlikely that a state will flip its vote.

The coinflip method produces a different winner on each run, and shows a high variance in the delegate count. This is because the votes for each state are nearly 50/50, so it is easy for states to 'flip' and change their vote. While this did give the simulation more randomness, it is less realistic than the percentage method, as each state's vote is extremely close to 50%. Potential improvements are discussed in the *Discussion Questions.*

The cost and time of the election remain unchanged regardless of the voting method used, and is unchanged for every run, as the cost and time of mailing is fixed. The results are displayed below.

Cost of Election	$2150719000
Duration of Election	10 Days

The two figures below display the results from a simulation of the election, using the political percentage method. In *figure 1* it is shown that the Red candidate won the majority of the states. It appears that the Red candidate was hugely more popular. However, the heatmap in *figure 2* shows that the results were much closer than the previous map showed. Most of the states are purple, indicating that, while the Red party did win the majority of the states, the vote counts within these states were usually quite close.

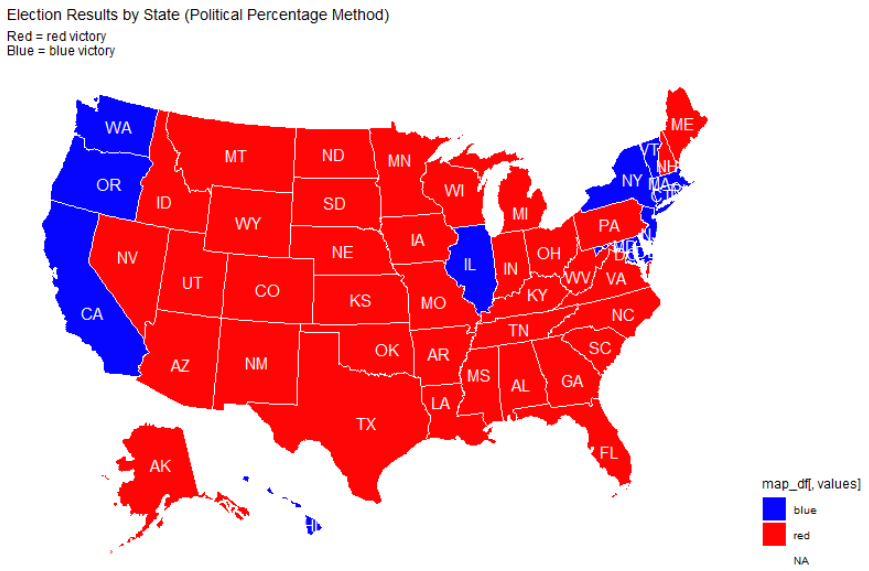

Figure 1:

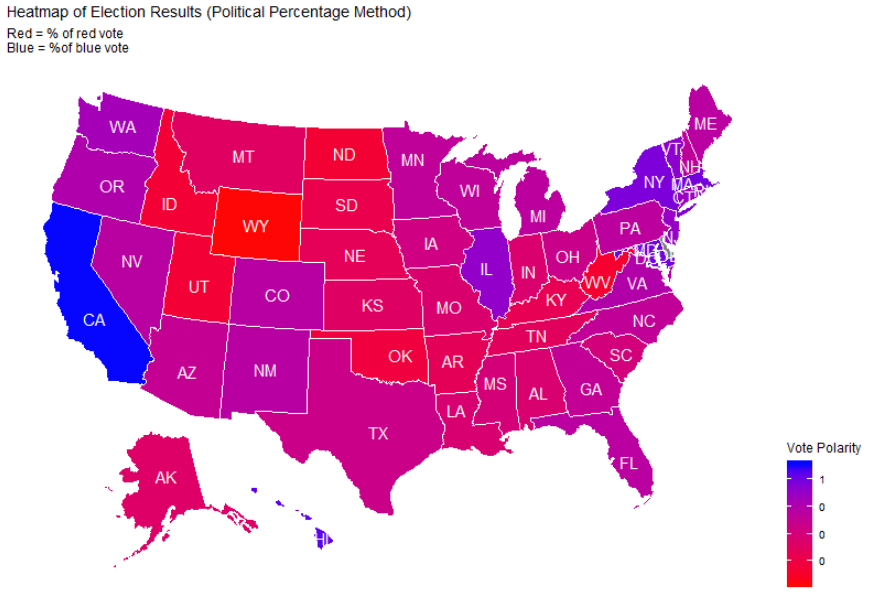

Figure 2: