Simulating a Large-Scale and Secure Voting System

Siddha Kilaru, Alex Riddle

Rensselaer Polytechnic Institute, Troy, NY

Abstract

This paper proposes an approach to simulate a large-scale and secure voting system. The process involves two main steps: generating encrypted votes and then tallying them up to determine the preferred candidate. The project leverages high-performance computing as the vote generation process can be distributed across multiple compute nodes to scale up voter counts. Moreover, the paper describes how the vote generation algorithm can be used in simulating election outcomes using Monte Carlo. Furthermore, the safeness of a fully digital voting system is also investigated in the paper. The paper explores the effectiveness of standard encryption and cryptography algorithm in ensuring voter privacy and integrity. We find that parallel computing massively helps in generating large amounts of voter data, aids in Monte Carlo simulations, and also provides a noticable speedup in tallying up votes. Additionally, we determine that **INCOMPLETE**.

Keywords: Distributed Computing, Voting Systems, Monte Carlo, Cryptography, Encryption Algorithms, Graph Algorithms, Algorithmic Game Theory

Contents

1. Introduction

In this paper, we want to simulate a large-scale and safe voting system. Our paper can be separated into two distinct steps. The first step is creating votes. In the real world, during an election day, a large population submits ballots containing rankings of their preferred candidates. We aim to simulate this by generating a file containing a list of candidate rankings equal to the number of voters. Note that in the real world, this large list of rankings is not localized to just one location (it would be spread across states, for example); however, for our model, we assume that all the votes have been gathered in the same place. After all the votes have been generated, they are then encrypted. The second stage of the experiment is tallying up the number of votes to find the preferred candidate. To do this, the encrypted votes must first be decrypted, and then in order to find the preferred candidate efficiently, we employ the Schulze method.

So how exactly does our project relate to high-performance computing? First, generating candidate listings is not exactly a trivial task. The population of the US is currently over 300 million, and if each voter is asked to rank their top 20 candidates, repeated generations can take time. In this paper, we develop a way to distribute this generation across multiple compute nodes in order to scale up our vote counts to large numbers. Moreover, we make use of the Fisher-Yates shuffle algorithm to efficiently generate unbiased permutations of candidate rankings. In addition to generating unbiased permutations, we experiment with biased permutations using custom distributions and make use of Monte Carlo simulations to see how candidate rankings are affected.

After the votes are generated, we use AES to encrypt the votes in blocks and distribute the blocks across compute nodes. AES is a block cipher, and block ciphers need to use modes to work securely. In particular, we are going to use the counter (CTR) mode of AES, which allows us to determine the key for any block of the large data, thus allowing us to parallelize the encryption and decryption process. Each node receives a chunk of the larger file, and each rank decrypts/encrypts the blocks in parallel using threads as well, which gives us two "levels" of parallelization for this process. We will also test using various key sizes of AES, starting with 128 and going up to 256. The larger the key size, the more expensive the encryption is, though the block sizes remain the same. After AES encryption, we then

Email addresses: kilars2@rpi.edu (Siddha Kilaru), riddla@rpi.edu (Alex Riddle)

hash (SHA-1) the encrypted votes in blocks and also spread the workload across compute nodes. The hashing is used to compare the received encryption to the original one to ensure no votes were modified.

Finally, the decrypted votes are tallied up. This tallying process requires creating a graph and running a graph algorithm, which we parallelize. The main advantage a high computing platform gives us is to scale voter numbers to extremely large numbers. Additionally, our parallel voting and encryption algorithms can serve as benchmarks for other supercomputing platforms as they are easily standardizable.

2. Generating Votes

2.1. A Simple Vote Model

A vote in our experiment consists of a preferential ordering of candidates. So given a set of candidates $C = \{c_1, c_2, c_3, \ldots, c_n\}$ where $n \geq 1$, we generate a permutation $P = \{c_1^*, c_2^*, c_3^*, \ldots, c_n^*\}$ of C according to some distribution. Moreover, for some for some $i < j$ where $i, j \in [1, n]$, c_i^* is strictly preferred to candidate c_j^*. For simplicity's sake, our model does not account for any ties, so if a voter equally prefers two or more candidates, we do not take that into account.

One of our main goals is to generate a number of votes $m \gg 1$ in very large quantities. Doing so allows us to run accurate Monte Carlo simulations and also test the robustness of our usage of SHA-1 and AES that create certain safety measures for the votes. The best way to do this is by storing the vote data in files rather than depending on program memory. There are two main reasons for this. First, storing the data in files is much more efficient for repeated access. Generating votes every time an executable starts is not practical and counterproductive. Second, storing the votes in files allow us to separate the processes of the voting process. That is, we use an executable to generate and encrypt votes into a file and then another executable to read in, decrypt votes, and generate a candidate ranking. Also, using files is much more convenient for repeated access.

2.2. Vote Generation Algorithm

To generate votes, we distribute the workload across several different compute nodes (ranks) using MPI. Also, we make use of MPI parallel I/O so each rank can write to a file in parallel as well. The main algorithm for this process is given below:

Algorithm 1 Parallel Vote Generation

```
1:  procedure VoteGen(rank, startId, endId)
2:      for i ← startId, endId do
3:          voteSequence.Append(RandomPermutation())
4:      end for
5:      size ← SizeOf(voteSequence)
6:      allSizes ← AllGather(size)          ▷ Sizes for all ranks.
7:      cursorPos ← CumulativeSum(AllSizes)
8:      MPIFileSeek(cursorPos[rank])
9:      MPIFileWriteAll(voteSequence)
10: end procedure
```

This algorithm is active in every single rank. The inputs to the algorithm are *rank*, *startId*, and *endId*. Different *startId*s and *endId*s are passed to the ranks which represent the section of votes each rank is responsible for generating and writing. They are calculated by partitioning the total number of voters across ranks and then carefully assigning the bounds for each rank. This partition can be calculated ahead of time because the total number of desired votes is passed to the executable as an argument. The first part of the algorithm is to generate the vote sequence. To do this, we loop across the rank's assigned range (prescribed by *startId* and *endId*) and append to *voteSequence*, which is a buffer that contains the vote sequences for all the assigned ids. Since the sizes of the buffer are not known until they are fully generated, the ranks need to share the buffer sizes with each other in order to write to the file in the correct place. Consequently, the next step is to accurately assign cursor positions for each rank; in parallel I/O, it is very important that alignment across different ranks is perfect; otherwise, you risk overwriting data. To achieve this, the sizes of the buffer are shared across all the ranks with each other and stored in *allSizes*. Using that array, we then generate the array *cursorPos*, which is just the cumulative sum. Finally, using *cursorPos*[*rank*], which is the correct cursor position for *rank*, the contents of the buffer (*voteSequence*) can be written to the file. Specifically, we use `MPI_File_seek` to move the file cursor and use the collective write operation `MPI_File_write_all` to write the contents of the buffer to the file:

```
MPI_File_seek(fh, cursor_pos[rank - 1],
              MPI_SEEK_SET);
MPI_File_write_all(fh, vote_sequence,
                   size, MPI_CHAR, NULL);
```

This is different than using independent I/O where every rank writes to a file independently; collective I/O mandates that all process are writing to a file at the same time, which allows MPI to perform certain optimization making it much more faster overall [5]. Moreover, by adjusting the cursor positions to distinct positions in the file allows for non-contiguos access, and this combined with collective I/O is one of the most efficient ways to concurrently write to a file [6].

One natural question to ask is if the parallel vote generation algorithm can be parallelized even further to achieve faster runtimes and generate more votes. The answer to that is yes, there is more oppurtunity for more parallelism in that algorithm, and it actually is fairly straightforward. The for loop from lines 2 to 4 can be parallelized because each random permutation is generated independent of one another. However, the fact that we are writing to a file means this not might not exactly be the case. We implemented a similar parallel algorithm in CUDA regarding random graph generation; however it ended up being much slower than just using MPI. The reason for this is because instead of keeping track of cursor positions outside the loop, cursor positions needs to be updated inside the loop. This critical constraint makes parallelizing the loop in lines 2-4 actually much more slower because it requires updating a shared variable writing to shared memory concurrently. However, once again, we brought this problem upon ourselves by making writ-

ing to a file a core rule. If we didn't have to write to a file, and could rely on program memory, then using CUDA and parallelizing the inner loop would be undoubtly faster.

2.3. Results

Figure 1 below shows the total time for generating a vote file containing $m = 10000$ votes, and $n = 20$ candidates for 1 up to 512 ranks. Moreover, the runtime is the result of running the full parallel vote generation algorithm.

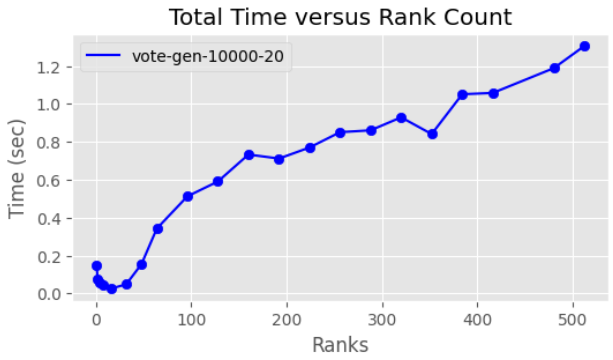

Figure 1: Compares the total runtime versus the number of ranks for running vote-gen on 10000 voters and 20 candidates. Each point represents an actual data point (access here) collected on AiMOS (DCS) supercomputer.

Interestingly, the runtime increases as the number of ranks increases, which is counter to what should happen; more compute power should result in a faster runtime. However, that is not always the case. The number of candidates is too small, so the overhead of coordinating a large number of ranks to write to a file ends up being much more costly than just a smaller number of ranks writing to a file. In fact, this can be observed in the figure: at first, the run time dips with, but as the number of ranks increases substantially, so does the runtime. Figure 2 below is still for $m = 10000$ and $n = 20$, but instead of showing the total runtime, the runtime for only generating permutations is shown (lines 2 to 4 of the parallel vote generation algorithm).

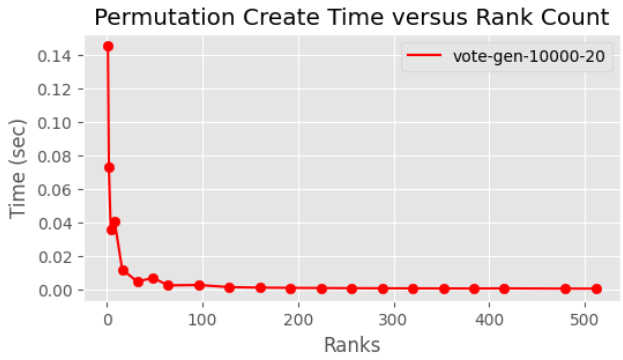

Figure 2: Compares the permutation generation runtime versus the number of ranks for running vote-gen on 10000 voters and 20 candidates. Each point represents an actual data point (access here) collected on AiMOS (DCS) supercomputer.

Unlike figure 1, there is a visible decrease in runtime as the number of ranks increase. This is because the figure does not include the runtimes for writing to the files, only generating the permutations, suggeting that it is very relatively expensive to collectively call MPI write operations for all ranks. However, as the number of candidates and voters increase, I expect the extra burden from additional coordination MPI ranks to be insignificant. In fact, looking at figure 3 below with $m = 10^8$ and $n = 20$, that exactly is the case.

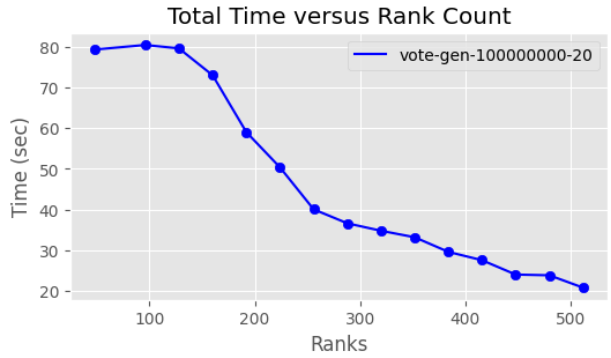

Figure 3: Compares the total runtime versus the number of ranks for running vote-gen on 10^8 voters and 20 candidates. Each point represents an actual data point (access here) collected on AiMOS (DCS) supercomputer.

Here the runtime does indeed decrease as the number of ranks increase. Without distributing the computing, we would not have been able to generate the file in a reasonable amount of time. Using only 1 rank was completely insufficient, and the batch job was cancelled over 10 minutes in. In fact, only until 48 ranks were we able to generate the file with a total runtime of about 3 minutes. Using 512 ranks reduced the runtime about 25% to 40 seconds.

To explore more how vote counts affect the figure below shows the relationship between vote count and runtime between a different number of ranks. As already explained above, a large number of ranks with a low vote count is counterproductive because the overhead cost is much higher than the I/O cost. Figure 4 belows provides some insight as to where this inflection point occurs number where the cost of overhead overturns I/O.

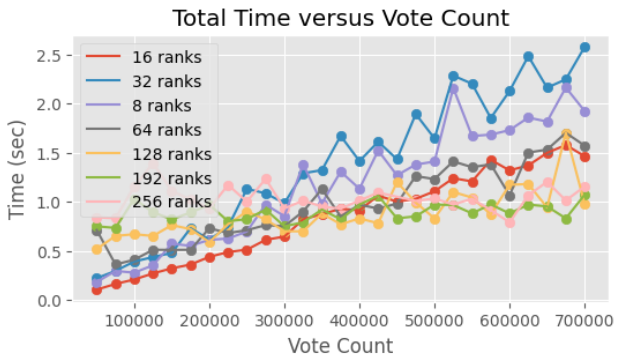

Figure 4: Compares the total runtime versus the number of votes. Each rank has its own line indicated by the legend. Each point represents an actual data point (access here) collected on AiMOS (DCS) supercomputer.

From this figure we can see that lower number rank counts perform much better up until where the vote count starts to reach 4×10^5. After that, higher number rank counts start to achieve a lower run time. One puzzling thing is that 8 and 16

ranks seem to perform much better than 32 ranks consistently for all vote counts; we redid this experiment to see if that was a fluke, but 32 ranks still performed much worse.

Figure 5 below shows the permutation creation time versus vote counts. As expected, indicated by figure 1, the higher rank count the permutation creation time will always be lower because there is no I/O time factored in. In the figure, this is immediately apparent with 256 ranks runtime consistently being lower than all the other ranks. However, again, one notable thing is that 32 ranks performs worse than expected, and the same observation was made after redoing the experiment.

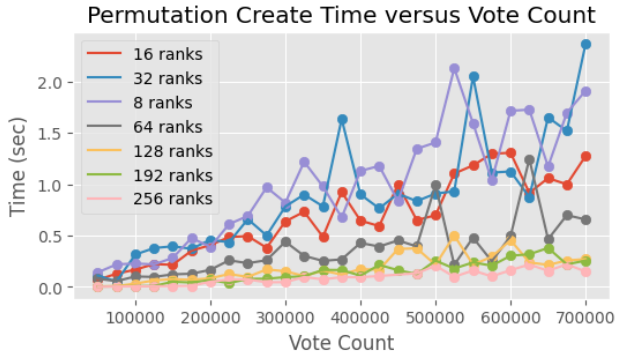

Figure 5: Compares the permutation creation time versus the number of votes. Each rank has its own line indicated by the legend. Each point represents an actual data point (access here) collected on AiMOS (DCS) supercomputer.

3. Tallying Votes

3.1. Schulze Method

The main algorithm we use to create a finalized candidate ranking is the Schulze method. The algorithm is a grpah algorithm, is similar to the Floyd-Warshall algorithm in that it compares paths between all pairs of vertices. As a result, the run time of the algorithm is $O(n^3)$, where n is the number of candidates [1]. This seems like an issue for scaling up; however the number of candidates stays constant and is typically a small number. As a result, their really is no use in parallelizing this aspect of the algorithm. Where parallelization comes into play is the actual creation of the graph, which we represent using an adjacency matrix. The algorithm is given below:

Algorithm 2 Preference Graph Generation

```
1:  procedure GraphGen(rank, startId, endId, data)
2:      for voter ← startId, endId do
3:          ranking = data[voter]        ▷ Get candidate ranking.
4:          for i ← 0, sizeOf(ranking) do
5:              for j ← i + 1, sizeOf(ranking) do
6:                  row ← ranking[i]
7:                  col ← ranking[j]
8:                  graph[row][col]+ = 1
9:              end for
10:         end for
11:     end for
12:     MPIReduce(graph, finalGraph, MPI_SUM)
13: end procedure
```

This algorithm is run on every rank concurrently. Similar to the first algorithm in section 2.2, the same arguments are passed in with the addition of *data*, which contains all the votes for each voter in the ranks id range. An example vote is in the form $\{c_1, c_2, c_3, \ldots, c_n\}$, which is passed to *ranking* on line 3. *graph* represents preferences for each candidate such that $graph[c_i][c_j] = k$ means that c_i is preffered to c_j a total of k times; note that $k \geq 0$. To generate *graph*, the following is done from lines 2 to 11: for all i and j where $i < j$, c_i is preferred to c_j, so we increment $graph[c_i][c_j]$ by 1. *graph* contains all the pairwise preferences only for the data stored on its own rank, so a reduce operation needs to be performed to sum up all the graphs. This is done on line 12 with `MPI_Reduce` using where $finalGraph$ is stored only on rank 0 and contains preferences for the whole vote data set:

```
MPI_Reduce(graph, final_graph,
           graph_size, MPI_INT,
           MPI_SUM, 0, MPI_COMM_WORLD);
```

After generating the final graph, the only step left is to calculate the final candidate ranking using Schulze method, which is fairly trivial.

3.2. Permutation Generation and Monte Carlo

A core part of a simulating a voting system is being able to generate a permutation of candidates (P), or a vote, according to some distribution that describes candidate placement relative to one another. P can be generated efficiently using the Fisher-Yates shuffle algorithm [2]; the run time of the algorithm is linear in the number of candidates, $O(|P|)$. The algorithm uniformly generates permuations such that given a complete set of $|P|!$ permutations $\mathbf{P} = \{P_1, P_2, P_3, \ldots, P_{|P|!}\}$ each P_i is equally likely to be generated with probability $1/|\mathbf{P}|$. Also, if you look at all the permutations in P, for some i and j where $i \neq j$, the amount of times c_i is preferred to c_j ($c_i >> c_j$) appears in exactly $|\mathbf{P}|/2$ different permutations. This means that

$$\mathbb{E}(c_i >> c_j) = \sum_{i=0}^{|\mathbf{P}|/2} \mathbb{P}(c_i >> c_j) = \frac{1}{2}.$$

This has some unique characteristics in regards to the preference graph. This means that every entry in the graph has an expected value of $m/2$, where m is the number of voters, regardless of the number of candidates. We can use Monte Carlo to verify this. Using $m = 1000000$ and $n = 2$ generates the preference graph:

$$\begin{bmatrix} 0 & 499315 \\ 500699 & 0 \end{bmatrix}$$

Using $m = 1000000$ and $n = 5$ generates the following preference graph:

$$\begin{bmatrix} 0 & 500440 & 499949 & 499807 & 500809 \\ 499568 & 0 & 499413 & 499470 & 500436 \\ 500059 & 500595 & 0 & 499703 & 500173 \\ 500201 & 500538 & 500305 & 0 & 500224 \\ 499199 & 499572 & 499835 & 499784 & 0 \end{bmatrix}$$

All the entries for both graphs are approximately $m/2$ and dividing all the entries by m approximately results in $\frac{1}{2}$.

Instead of uniformly sampling permutations from **P**, we can apply different distributions depending on how likely we think one candidate is going to be preferred to another. Consider the distribution where a candidate c_i is unconditionally preferred to other candidates. That is for a single fixed candidate c^*, $\mathbb{P}(c^* >> c_j) = 1$, and consequently, $\mathbb{E}(c^* >> c_j) = 1$. For all i and j where $i \neq j$ and $c_i \neq c^*$, $\mathbb{E}(c^* >> c_j) = \frac{1}{2}$. Using $m = 1000000$, $n = 5$ and $c^* = 3$ generates the following prefence graph and candidate ranking:

$$\begin{bmatrix} 0 & 500147 & 0 & 499580 & 500224 \\ 499861 & 0 & 0 & 499935 & 499113 \\ 1000008 & 1000008 & 0 & 1000008 & 1000008 \\ 500428 & 500073 & 0 & 0499800 & \\ 499784 & 500895 & 0 & 500208 & 0 \end{bmatrix}$$

Ranking: **3**, 4, 1, 5, 2

As you can see, all the entries except the third row are approximately equal $m/2$, the third row is approximately equal to m, and candidate 3 ranks first.

The two distributions shown in this section are relatively simple, and their expected values can easily be computed. However, as the number of candidates increase, and if the distributions are not exactly trivial, Monte Carlo simulations may be needed to get preference statistics.

4. Encryptin' and Hashin' Votes

5. Project Organization

6. Summary and Future Work

References

[1] Schulze, M. (2018). The Schulze method of voting. arXiv preprint arXiv:1804.02973.

[2] Hazra, T. K., Ghosh, R., Kumar, S., Dutta, S., & Chakraborty, A. K. (2015, October). File encryption using fisher-yates shuffle. In 2015 International Conference and Workshop on Computing and Communication (IEMCON) (pp. 1-7). IEEE.

[3] Schulze, M. (2011). A new monotonic, clone-independent, reversal symmetric, and condorcet-consistent single-winner election method. Social choice and Welfare, 36(2), 267-303.

[4] Corbett, P., Feitelson, D., Fineberg, S., Hsu, Y., Nitzberg, B., Prost, J. P., Wong, P. (1996). Overview of the MPI-IO parallel I/O interface. Input/Output in Parallel and Distributed Computer Systems, 127-146.

[5] Pacheco, P. (1997). Parallel programming with MPI. Morgan Kaufmann.

[6] Ching, A., Choudhary, A., Coloma, K., Liao, W. K., Ross, R., and Gropp, W. (2003, May). Noncontiguous i/o accesses through mpi-io. In CCGrid 2003. 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid, 2003. Proceedings. (pp. 104-111). IEEE.

[10] Leslie Lamport (1994) *LaTeX: a document preparation system*, Addison Wesley, Massachusetts, 2nd ed.

[10] Leslie Lamport (1994) *LaTeX: a document preparation system*, Addison Wesley, Massachusetts, 2nd ed.

[10] Leslie Lamport (1994) *LaTeX: a document preparation system*, Addison Wesley, Massachusetts, 2nd ed.

[10] Leslie Lamport (1994) *LaTeX: a document preparation system*, Addison Wesley, Massachusetts, 2nd ed.