CIE 564

Project Report

Using MPI for Faster Analysis of Elections Data

A Project Done By:

Shrouk Mohamed - 201401023

Ahmed Wael - 201500862

Instructor’s Name:

Dr. Anas

Eng. Hanan

Table of Contents:

Introduction & Problem Statement

At certain times when we have a large load of data coming from a certain source, we would like to have a fast analysis on this data, an example of this situation is finding out the results of elections, which is the problem we were trying to work on.
We would like to build a solution which uses Message Passing library in order to do a faster analysis on elections and find out the winning candidate of the elections by going through the list of all of the voters preferences.
The main idea is to have a certain number of running processes among which the list of all of the voter's preferences are distributed, after each process does the required analysis on its part of the data, the partial solutions are all sent to the master process in order to be aggregated to calculate the final solution.

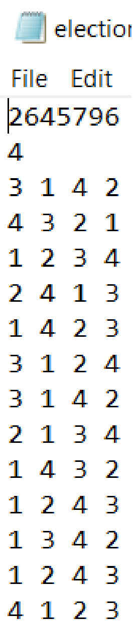
Phase 1 requirements

1st: Generate the preferences list file:

In this part, we generate a file which contains the following:

1. Generate the number of candidates who will be participating in the elections process
2. Generate the number of voters upon which the results will be based
3. Generate a preference list for each of the voters that goes from the most preferred candidate to the least preferred one

The following screenshot is an example of the output of the file generation part:



```
2645796
4
3 1 4 2
4 3 2 1
1 2 3 4
2 4 1 3
1 4 2 3
3 1 2 4
3 1 4 2
2 1 3 4
1 4 3 2
1 2 4 3
1 3 4 2
1 2 4 3
4 1 2 3
```

This file is passed on to the MPI program as an input file in order to generate the results of the elections process

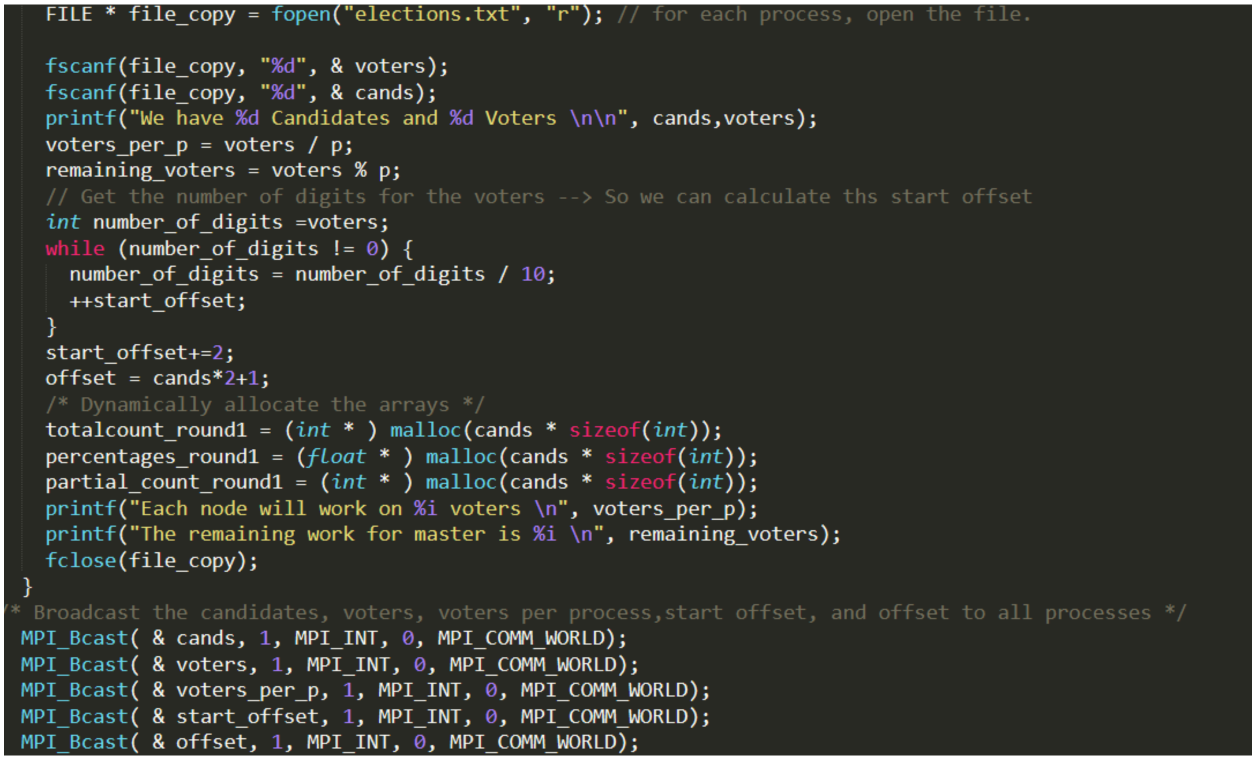
2nd: Read the input file into the program:

In this part, the file generated from the first part is passed on to the MPI program as an input, but the file is too large because of the huge number of voters and we do not want to read it at once.

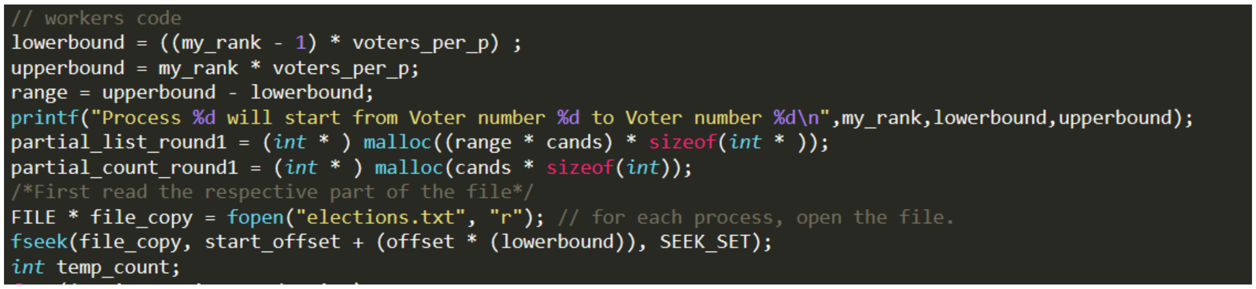
First, we would extract from the file the number of candidates and number of voters.

After that we set the proper initial offset to read from the file, and each process will increment to this initial offset according to its rank.

The following code snippet shows the part of the code where this is defined:



```c
    FILE * file_copy = fopen("elections.txt", "r"); // for each process, open the file.

    fscanf(file_copy, "%d", & voters);
    fscanf(file_copy, "%d", & cands);
    printf("We have %d Candidates and %d Voters \n\n", cands,voters);
    voters_per_p = voters / p;
    remaining_voters = voters % p;
    // Get the number of digits for the voters --> So we can calculate ths start offset
    int number_of_digits =voters;
    while (number_of_digits != 0) {
      number_of_digits = number_of_digits / 10;
      ++start_offset;
    }
    start_offset+=2;
    offset = cands*2+1;
    /* Dynamically allocate the arrays */
    totalcount_round1 = (int * ) malloc(cands * sizeof(int));
    percentages_round1 = (float * ) malloc(cands * sizeof(int));
    partial_count_round1 = (int * ) malloc(cands * sizeof(int));
    printf("Each node will work on %i voters \n", voters_per_p);
    printf("The remaining work for master is %i \n", remaining_voters);
    fclose(file_copy);
  }
/* Broadcast the candidates, voters, voters per process,start offset, and offset to all processes */
  MPI_Bcast( & cands, 1, MPI_INT, 0, MPI_COMM_WORLD);
  MPI_Bcast( & voters, 1, MPI_INT, 0, MPI_COMM_WORLD);
  MPI_Bcast( & voters_per_p, 1, MPI_INT, 0, MPI_COMM_WORLD);
  MPI_Bcast( & start_offset, 1, MPI_INT, 0, MPI_COMM_WORLD);
  MPI_Bcast( & offset, 1, MPI_INT, 0, MPI_COMM_WORLD);
```



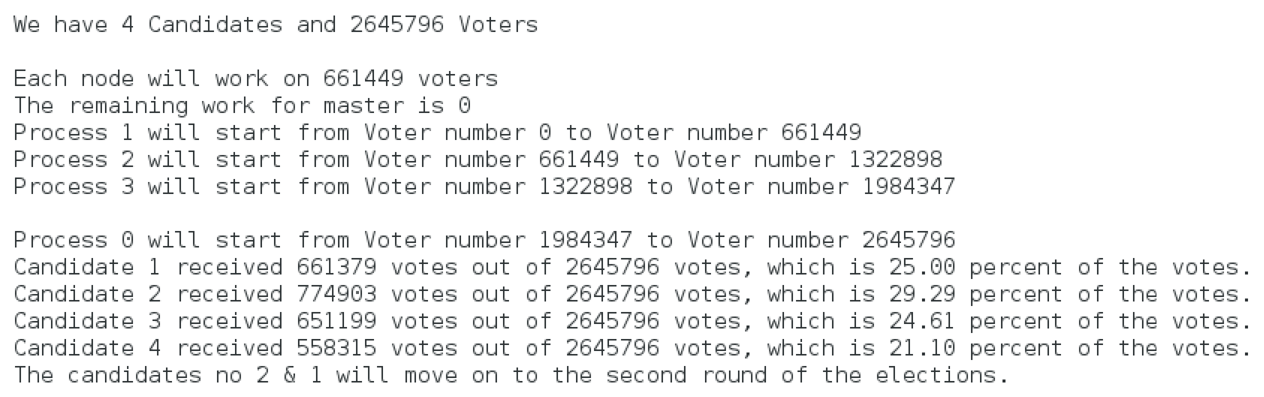
```c
// workers code
lowerbound = ((my_rank - 1) * voters_per_p) ;
upperbound = my_rank * voters_per_p;
range = upperbound - lowerbound;
printf("Process %d will start from Voter number %d to Voter number %d\n",my_rank,lowerbound,upperbound);
partial_list_round1 = (int * ) malloc((range * cands) * sizeof(int * ));
partial_count_round1 = (int * ) malloc(cands * sizeof(int));
/*First read the respective part of the file*/
FILE * file_copy = fopen("elections.txt", "r"); // for each process, open the file.
fseek(file_copy, start_offset + (offset * (lowerbound)), SEEK_SET);
int temp_count;
```

The master process is responsible for reading the number of candidates and the number of voters from the file, then divide the number of voters by the number of processes to know how many voters each of the processes will handle.

The master process will also be responsible for handling the remaining number of voters which could not be divided among the processes.

Dynamic allocation is used in order to form the 2-D array which contains all the preferences lists of all voters, this is better because the size of this array would be too big if the program is run with only one or two processes, so dynamic allocation handles this problem.

After each process reads its own portion of voters' preferences into its 2D array, it would try to find the fraction of votes each of the candidates in the list got as a most preferred candidates, a 1-D array called Partial_Count of length C, no of candidates, stores this count for each of the candidates.
All of the processes would send out this array to the master process using the function MPI_SEND(), the master process receives all of the Partial_Count arrays using MPI_RECEIVE(), then aggregates all of these partial count arrays into one Total_Count array, and then calculate the percentage of votes for each of the candidates during the first round and displays that on the console.
After that a decision needs to be made for whether this election process will need a second round or not, the master process would search for whether any of the candidates has received more than 50% of the votes, if this condition is met, the master process would print out to the console the number of the winning candidate, that he has won the elections and that there is no need to go through a second round of elections, else, the master process would search for the two candidates that have received the most amount of votes, and display that these two candidates are going to move on to the second round of elections.
The numbers of the two candidates who will move on to the second round of elections are stored in the variables cand1r2 and cand2r2.
The following screenshot displays an example of the code output for the first round of elections:



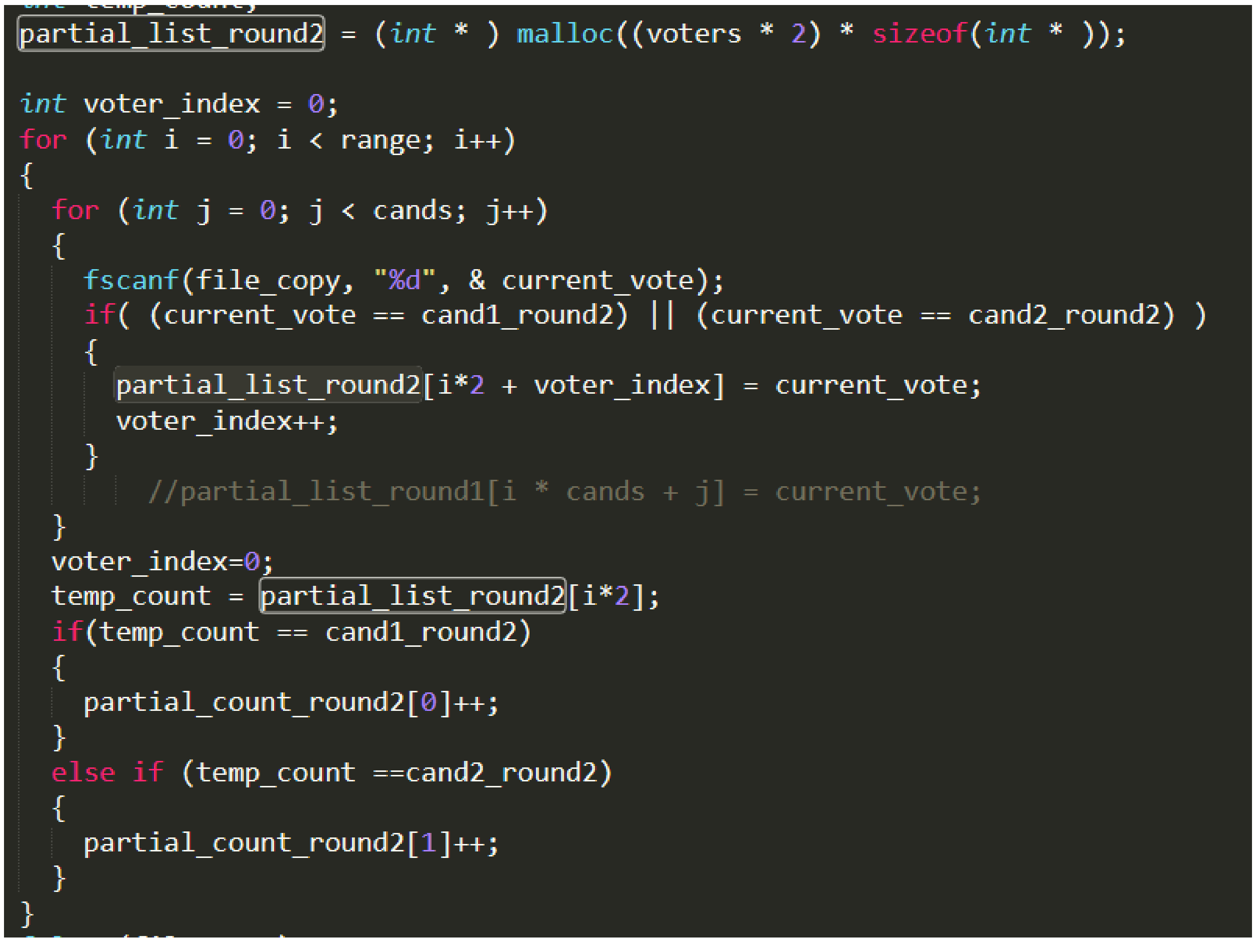
```
We have 4 Candidates and 2645796 Voters

Each node will work on 661449 voters
The remaining work for master is 0
Process 1 will start from Voter number 0 to Voter number 661449
Process 2 will start from Voter number 661449 to Voter number 1322898
Process 3 will start from Voter number 1322898 to Voter number 1984347

Process 0 will start from Voter number 1984347 to Voter number 2645796
Candidate 1 received 661379 votes out of 2645796 votes, which is 25.00 percent of the votes.
Candidate 2 received 774903 votes out of 2645796 votes, which is 29.29 percent of the votes.
Candidate 3 received 651199 votes out of 2645796 votes, which is 24.61 percent of the votes.
Candidate 4 received 558315 votes out of 2645796 votes, which is 21.10 percent of the votes.
The candidates no 2 & 1 will move on to the second round of the elections.
```

Phase 2 requirements

This part is about handling the case when no one of the candidates receives more than 50% of the votes during the first round of elections, another round of elections is going to run between the two candidates who have received the most amount of votes during the first round, the following are the steps followed in the code in order to handle this round.

In this round each of the processes is responsible for handling the same portion of the input file as in the first round, however during this round each of the processes filters out only the candidates involved in the second round from its list of voters, so the size of the 2D array for each process now is (2*No_Of_Voters_Per_Process).

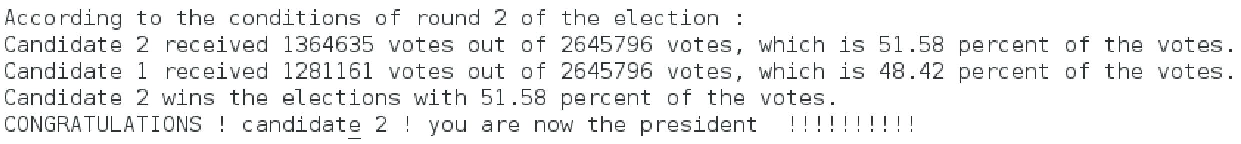
The following screenshot shows an example of the array formed by each of the processes during this phase:



```
partial_list_round2 = (int * ) malloc((voters * 2) * sizeof(int * ));

int voter_index = 0;
for (int i = 0; i < range; i++)
{
  for (int j = 0; j < cands; j++)
  {
    fscanf(file_copy, "%d", & current_vote);
    if( (current_vote == cand1_round2) || (current_vote == cand2_round2) )
    {
      partial_list_round2[i*2 + voter_index] = current_vote;
      voter_index++;
    }
      //partial_list_round1[i * cands + j] = current_vote;
  }
  voter_index=0;
  temp_count = partial_list_round2[i*2];
  if(temp_count == cand1_round2)
  {
    partial_count_round2[0]++;
  }
  else if (temp_count ==cand2_round2)
  {
    partial_count_round2[1]++;
  }
}
```

After that the same technique as the first round is used in order to find out the winner for the second round, i.e. calculating partial counts, sending to the master process, aggregating the count, deciding on the winner.

The following screenshot shows the final output of the MPI program:



```
According to the conditions of round 2 of the election :
Candidate 2 received 1364635 votes out of 2645796 votes, which is 51.58 percent of the votes.
Candidate 1 received 1281161 votes out of 2645796 votes, which is 48.42 percent of the votes.
Candidate 2 wins the elections with 51.58 percent of the votes.
CONGRATULATIONS ! candidate 2 ! you are now the president  !!!!!!!!!!
```

Handling working only with one process was also done correctly.