Adeniyi Adeboye CIS 677 – Project 4

MPI for Genomics: Genome-wide Association Study (GWAS)

Problem Description

The aim of this project is to develop an MPI-based message-passing application/program that analyses raw microarray data and identifies top differentially expressed genes between the **Renal and Control groups** of patient samples.

To determine these top differentially expressed genes signatures, **A discriminant algorithm** is developed based on the **T-statistic value (t-Test)** for each gene between the Renal and Control group. The **student's t-statistic value** is based on this formula:

$$\frac{\mu_1 - \mu_2}{\sqrt{\left(\frac{\sigma_1^2}{n_1}\right) + \left(\frac{\sigma_2^2}{n_2}\right)}}$$

where μ_1, μ_2 are the means of the two groups i.e., Renal and Control, σ_1^2, σ_2^2 are the sample standard deviation of the two groups, and n1 and n2 are the number of samples in the two groups for each genes. All together the above formula is used to calculate the t-stat value for each gene.

Now, to ***calculate or get the true significance of the resulting t-statistic value*** of each gene, we would need to compare the t-statistic value of each gene to a distribution of t-statistics generated by randomly permuting the sample groupings for each gene a number of times.

The random generated t-statistic distribution is done for each gene, and from this distribution we obtain the mean and standard deviation, and the significance of a discriminating gene can be determined by comparing its t-statistic to the distribution of t-statistics generated by a random permutation of samples. If the initial t-statistic of the gene is ≥ 3 standard deviations from the mean of the distribution formed using random sample permutations, then with 97% confidence that gene is a discriminator.

Each gene is ranked by its discriminant score (D) based on this formula:

$$\frac{|t_s - \mu_D|}{\sigma_D}$$

where t_s is T-statistic of a gene, μ_D is the mean of the mean of the distribution of random sample permutations and σ_D is the standard deviation of the random permutation distribution.

A gene's D-score is equivalent to the actual t-statistic's z-score with respect to a distribution of random t-statistics.

Methods

To compute the T-test value for each gene and subsequently calculates its discriminant score we develop two different programs one that is **Sequential** and another one that is based on **MPI Parallel computation.**

NOTE: The sample size of the whole microarray data is 4532 (i.e. 4532 Genes) excluding 8 Genes with less than 3 observations/patients for each group (i.e. either Renal or Control) for both Sequential and MPI Parallel.

Major steps taken in execution of sequential and MPI Parallel programs include:

1. Reading the data from csv files into 2D Vectors (separating treading in the Renal, Control and Gene Index csv files)

2. Calculating the T-Test value for each Gene Index

3. Computing the random permutations (1000 & 5000) for each Gene Index based on its number of samples for both Renal and Control groups.

4. Calculating the D-Score for each Gene Index based on the random permutation and its initial t-test value.

5. Sort the Gene Indices based on their D-Score

6. Write the resulting vector of D Score and Gene Indices into csv file.

MPI Parallel Computation

For the MPI Parallel computation, I basically used ***MPI_Scatter(), MPI_Barrier() and MPI_Gather()*** for execution.

So, we start by splitting the data into i.e. Scatter (***using MPI_Scatter()***) into the number of processes we want to use, for this project I used ***1, 2, 4, 11, 22, and 44 number of*** **processes**, this is because with this number of processes I **EQUALLY** split/Scatter the data into all available processes. This means only a part of the data (i.e., certain number of rows) are assigned to each process to execute.

4532/1 = 4532 – One process gets 4532 samples

4532/2 = 2666 – Two processes get 2666 samples

4532/4 = 1133 – Four processes get 1133 samples

4532/11 = 412 – Eleven processes get 412 samples

4532/22 = 206 – Twenty-two processes get 206 samples

4532/44 = 103 – Forty-four processes get 103 samples

Then, I used ***MPI_Barrier()*** to synchronize all the data into their respective processes, so that all portions of the data are equally delivered to the respective processes before moving forward.

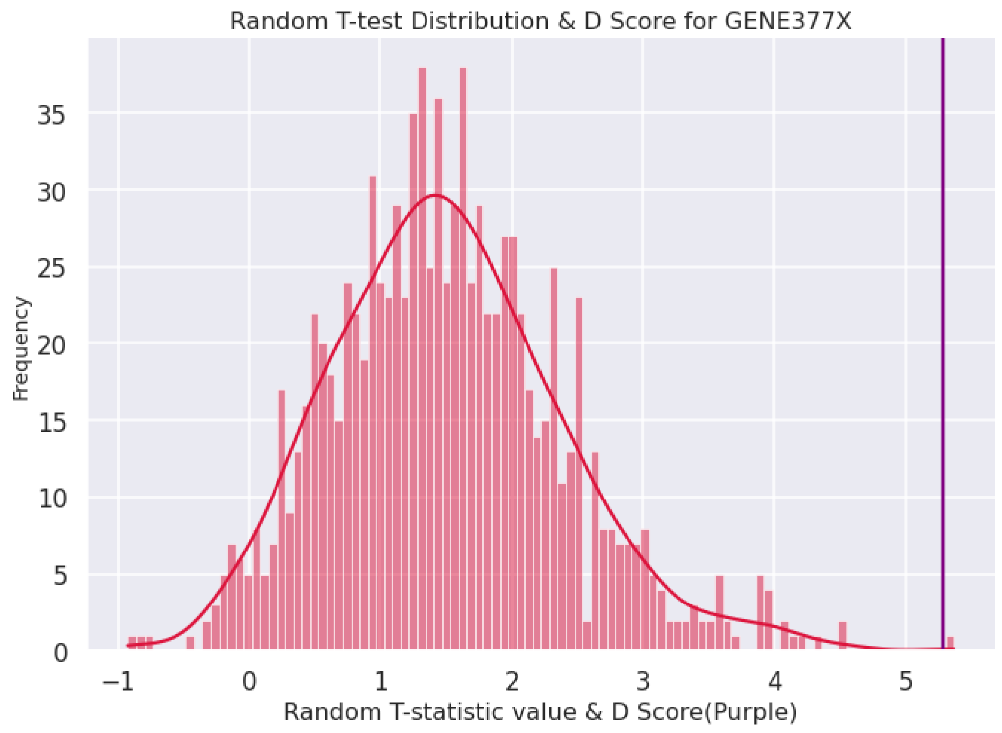
Thereafter, I computed all the necessary discriminant calculations i.e., calculate t-test, obtain random permutations for each gene, and calculate the D-Score based on random permutation and initial t-test value.

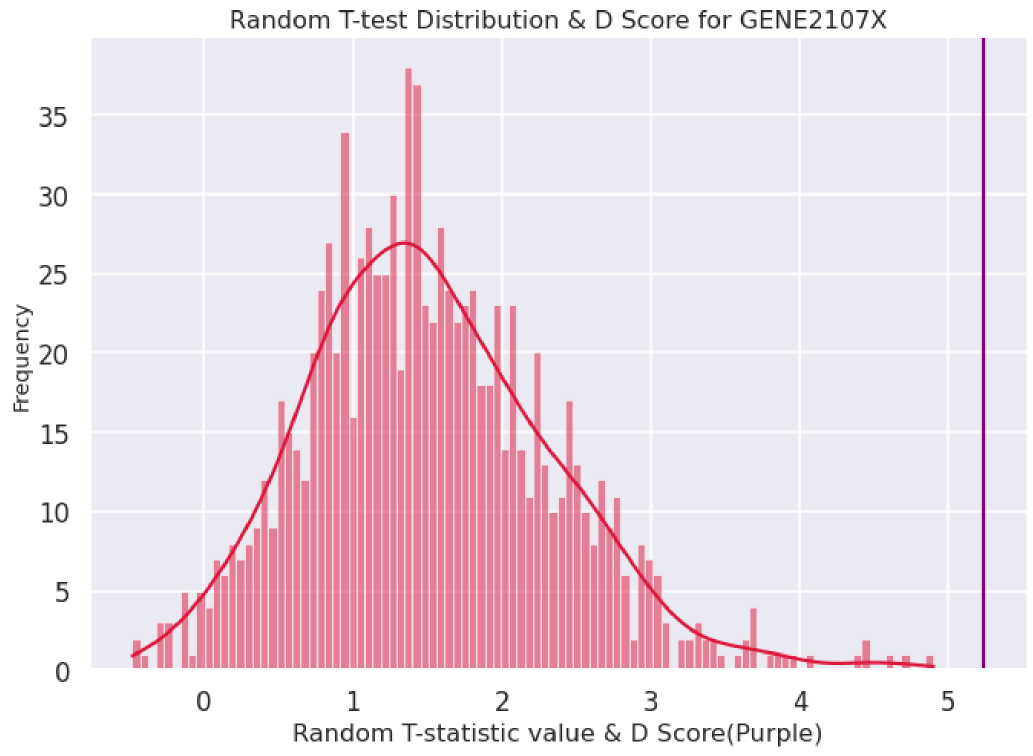
After all calculations, then I called ***MPI_Gather()*** to gather the results of the D-Scores of each gene index from all different processes used to a single D-Score vector back to the root process.

The sorting of the D-Score and writing to Csv file the results were done in the root process outside of the Scatter and Gather to allow consistency of the data.

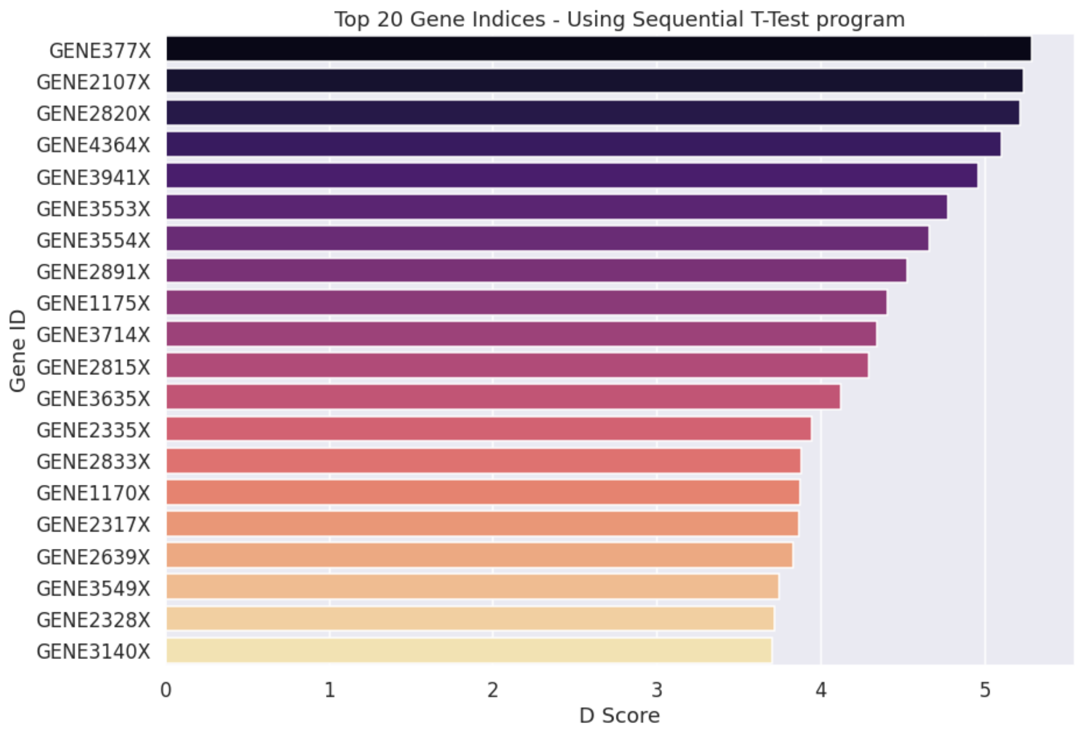
Results

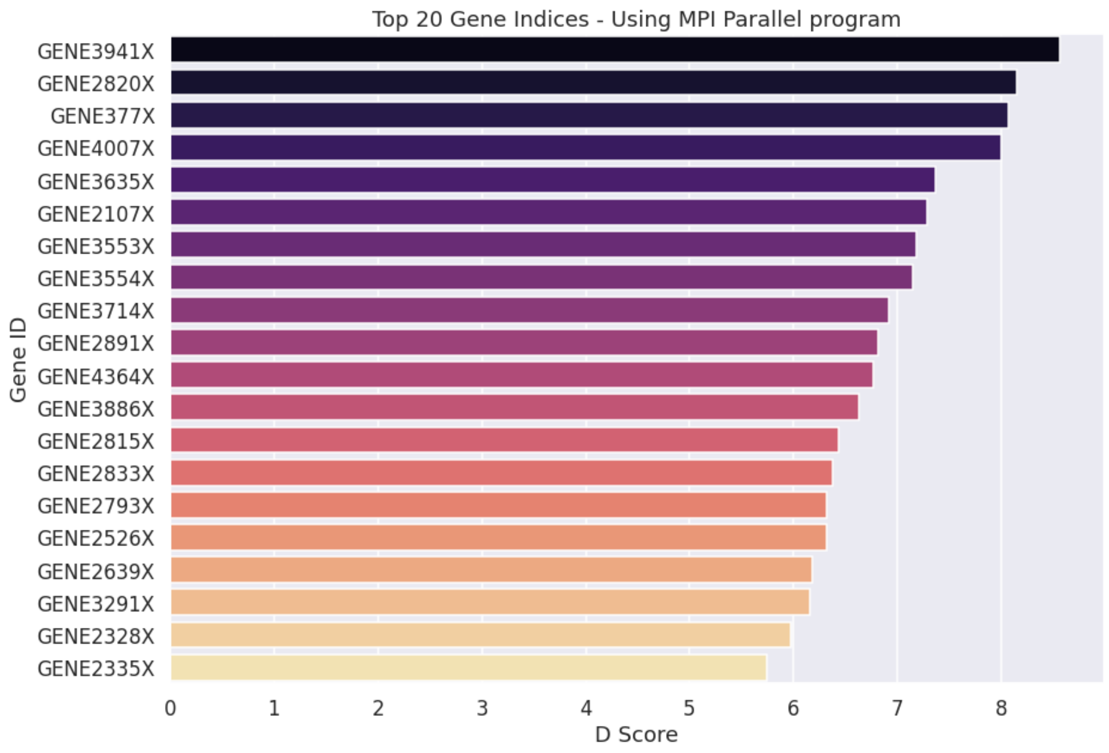
Random T-Test distribution and D scores of some of the top discriminant genes *i.e., **GENE377X and GENE2107X***:





Top 20 Discriminant Genes from both Sequential and MPI Parallel Programs:



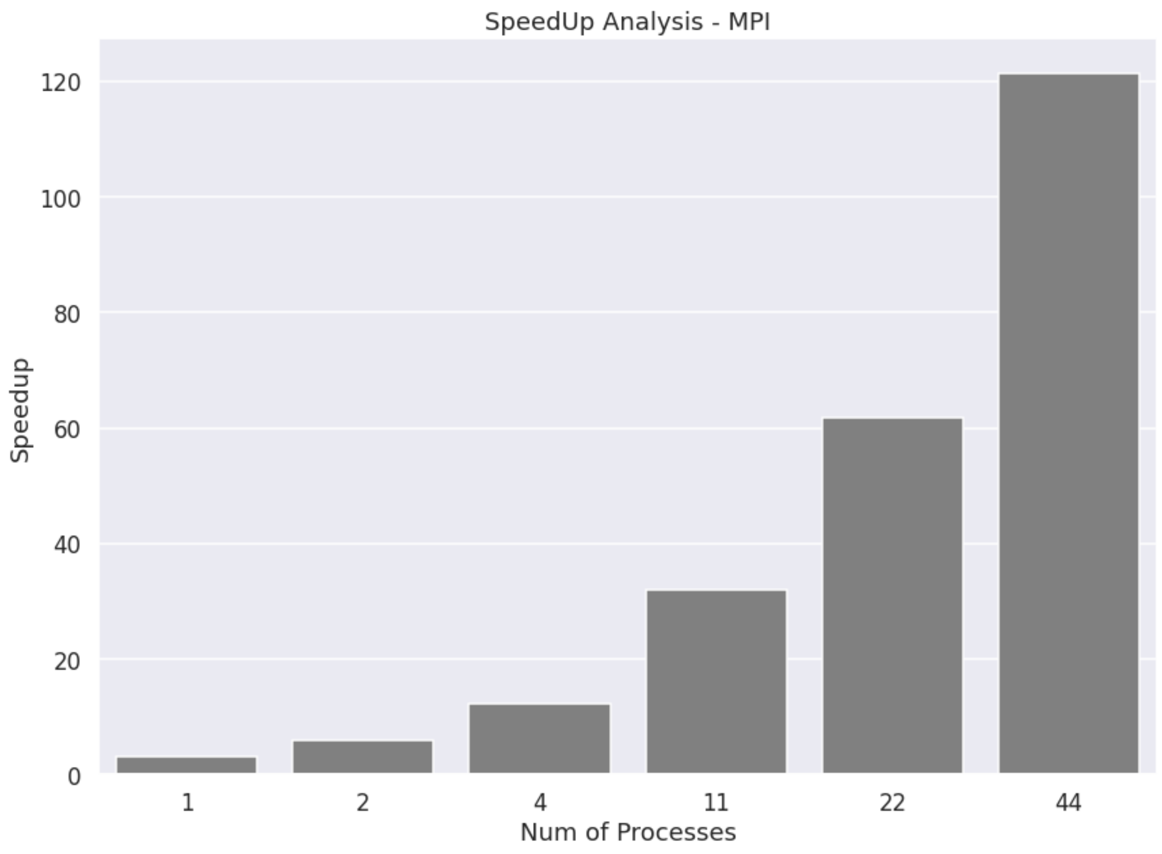


Based on the two above figures of the top 20 Gene Indices, we observed that the D-Scores for the sequential and MPI Parallel programs are not the same, this maybe due to the random permutations done separately on the two programs at different times and the way the two programs were executed.

One thing that is common to both programs, is that some Genes such as ***GENE 377X, GENE 2820X, GENE 2107X, GENE 4364X, GENE 3941X, GENE 3553X*** to mention but a few, are present in the top 10 discriminant genes in both programs, this indicates that these above-mentioned genes are said to have significant/true signatures (***true discriminators***) that can be used to differentially separate Renal groupings from the Controls.

Speed Up

In terms of speed up, we see high speed up (***i.e., > 100X***) with increasing number of processes based on the below figure:

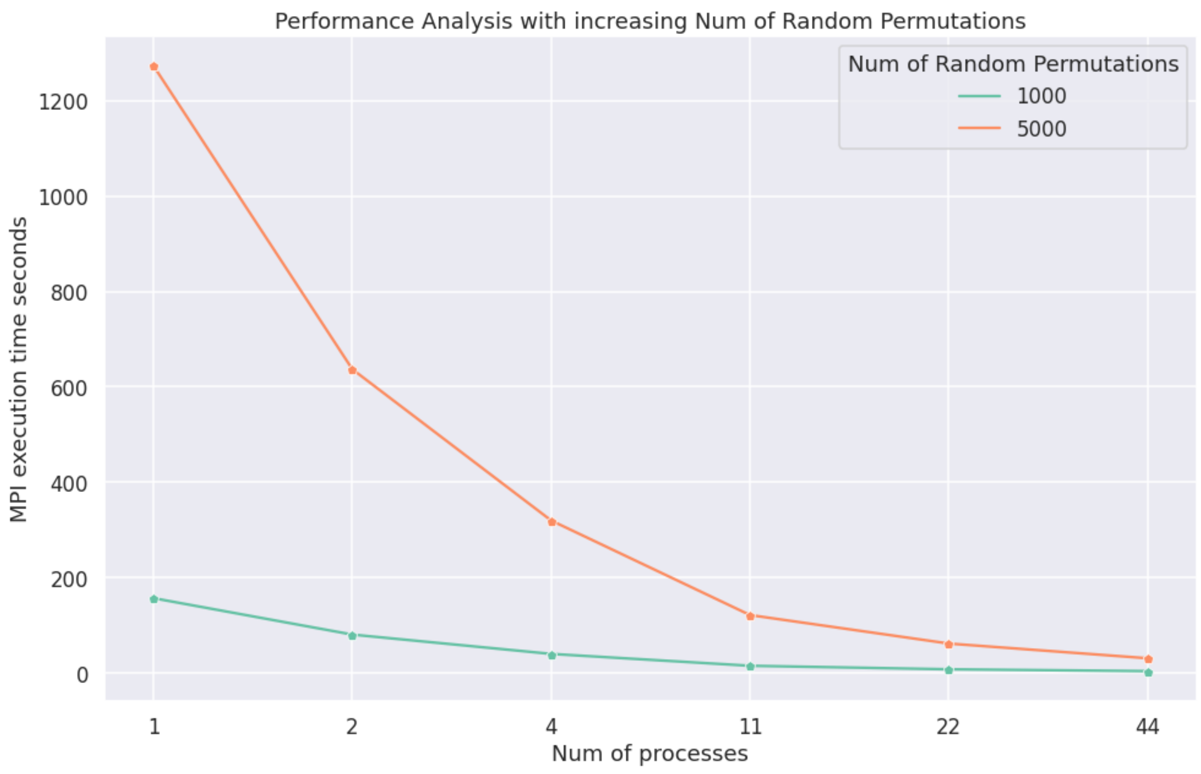


As observed, we observed HUGE Speed up with increasing processes i.e.

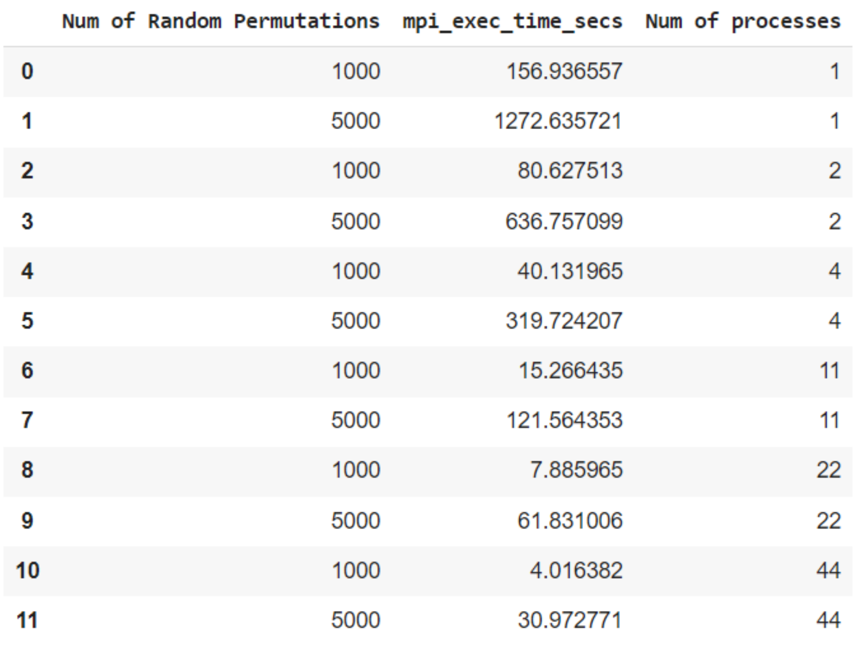
	process	speed_up
0	1	3.109537
1	2	6.052525
2	4	12.159883
3	11	31.965551
4	22	61.882091
5	44	121.502387

Performance Analysis

For the performance analysis, I used 1000 and 5000 random permutations for each gene to see how computationally expensive is going to be, when we increased the number of random permutations.



As observed from the above figure, with increase in random permutations, the MPI parallel execution time exponentially increase, for example for 2 processes the execution time went from approx. 80 seconds for 1000 random permutations to 636 seconds for 5000 random permutations, which overall makes this discriminant analysis based on random permutation computational expensive.



	Num of Random Permutations	mpi_exec_time_secs	Num of processes
0	1000	156.936557	1
1	5000	1272.635721	1
2	1000	80.627513	2
3	5000	636.757099	2
4	1000	40.131965	4
5	5000	319.724207	4
6	1000	15.266435	11
7	5000	121.564353	11
8	1000	7.885965	22
9	5000	61.831006	22
10	1000	4.016382	44
11	5000	30.972771	44

Conclusion

Overall, we see HUGE speed up (i.e., ***greater than 100X speedup***) when we use MPI Parallel program to compute this discriminant analysis, and it would be recommended to use MPI (***most especially Scatter and Gather***) for this kind of computation, as it distributes the samples equally among the available processes, which saves time.

However, with increase in the number of random permutations, we see huge increase in execution time of the MPI programs, which basically ***place a limit*** on the number of random permutations that could be done because it is computationally expensive.

References:

https://mpitutorial.com/tutorials/mpi-broadcast-and-collective-communication/

https://mpitutorial.com/tutorials/mpi-scatter-gather-and-allgather/

Sequential Code:

```
#include <iostream>     // cout, endl
#include <fstream>      // fstream
#include <sstream>
#include <vector>
#include <string>
#include <algorithm>    // copy
#include <iterator>     // ostream_operator
#include <random>
#include <tuple>
#include <utility>
#include <cmath>
#include <numeric>      // std::iota
#include <algorithm>
#include <chrono>

using namespace std;

template<typename T>
std::vector<T> flatten(const std::vector<std::vector<T>>& orig)
{
   std::vector<T> ret;
```

```
    for (const auto& v : orig)
        ret.insert(ret.end(), v.begin(), v.end());
    return ret;
}

template <typename T>
vector<size_t> sort_indexes(const vector<T>& v) {

    // initialize original index locations
    vector<size_t> idx(v.size());
    iota(idx.begin(), idx.end(), 0);

    // sort indexes based on comparing values in v
    // using std::stable_sort instead of std::sort
    // to avoid unnecessary index re-orderings
    // when v contains elements of equal values
    stable_sort(idx.begin(), idx.end(),
        [&v](size_t i1, size_t i2) {return v[i1] < v[i2]; });

    return idx;
}

vector<vector<string>> parseCsvToVector(string fileName) {
    std::ifstream csv(fileName);
    std::string line;
    std::vector <std::vector<std::string>> matRows;

    if (csv.is_open()) {
        for (std::string row_line; std::getline(csv, row_line);)
        {
            matRows.emplace_back();
            std::istringstream row_stream(row_line);
            for (std::string column; std::getline(row_stream, column, ',');)
                matRows.back().push_back(column);
        }
        csv.close();
    }
    else {
        cout << "Unable to open file";
    }

    /*for (size_t i = 0; i < matRows.size(); i++) {
        for (size_t j = 0; j < matRows[i].size(); j++) {
            cout << matRows[i][j] << " ";
        }
        cout << endl;
    }*/
    int Nrows = matRows.size();
    int Ncols = matRows[0].size();

    cout << "number of rows: " << Nrows << endl;
    cout << "number of columns: " << Ncols << endl;
    return matRows;

}

void write_csv(vector<vector<string>> dataset, string filename) {

    // Create an output filestream object
    std::ofstream myFile(filename);

    // Send data to the stream
    for (int i = 0; i < dataset.size(); ++i)
    {
        for (int j = 0; j < dataset[i].size(); ++j)
        {
            myFile << dataset[i][j];
            if (j != dataset[i].size() - 1) myFile << ","; // No comma at end of line
        }
        myFile << "\n";
    }

    // Close the file
    myFile.close();
    cout << "Successfully written data into " << filename << endl;
}



vector<vector<float>> return2DFloatVector(vector<vector<string>> vec) {
    vector< vector<float > > matFloat;

    for (size_t i = 0; i < vec.size(); ++i)
    {
        matFloat.push_back(std::vector<float>());
        for (size_t j = 0; j < vec[i].size(); j++) {
            string s = vec[i][j];
            string chars = "";
            for (char c : chars) {
                s.erase(std::remove(s.begin(), s.end(), c), s.end());
            }
            if ((!s.empty()) & (s.size() > 1)) {
                matFloat[i].push_back(stof(s));
                //cout << matRows[i][j] << " ";
            }
        }
        //cout << endl;
    }

    int Nrows = matFloat.size();
    int Ncols = matFloat[0].size();
    cout << "number of rows: " << Nrows << endl;
    cout << "number of columns: " << Ncols << endl;
    return matFloat;

}

vector<vector<string>> return2DStringVector(vector<vector<float>> vec) {
    vector< vector<string > > matString;

    for (size_t i = 0; i < vec.size(); ++i)
    {
        matString.push_back(std::vector<string>());
        for (size_t j = 0; j < vec[i].size(); j++) {
            matString[i].push_back(to_string(vec[i][j]));
            //cout << matRows[i][j] << " ";
        }
        //cout << endl;
    }

    int Nrows = matString.size();
    int Ncols = matString[0].size();
    cout << "number of rows: " << Nrows << endl;
    cout << "number of columns: " << Ncols << endl;
    return matString;

}

tuple<vector<vector<float>>,    vector<vector<float>>,    vector<string>>    returnVecwithKsize(vector<vector<float>>
disVec,
    vector<vector<float>> conVec, vector<string> geneIdVec, int k) {

    for (size_t i = 0; i < disVec.size(); ++i)
    {
        vector<float> myVec1, myVec2;
        for (size_t j = 0; j < disVec[i].size(); j++) {
            myVec1.push_back(disVec[i][j]);
        }
        for (size_t l = 0; l < conVec[i].size(); l++) {
            myVec2.push_back(conVec[i][l]);
        }
        if ((myVec1.size() < k) | (myVec2.size() < k)) {
            disVec.erase(disVec.begin()+i);
            conVec.erase(conVec.begin()+i);
            geneIdVec.erase(geneIdVec.begin()+i);
        }
        myVec1.clear();
        myVec2.clear();
    }

    cout << "number of Diesease rows: " << disVec.size() << endl;
    cout << "number of Control rows: " << conVec.size() << endl;
    cout << "number of Gene ID rows: " << geneIdVec.size() << endl;

    return { disVec, conVec, geneIdVec };

}

// Function to find mean.
float Mean(vector<float> vec)
{
    float sum = 0;
    for (auto& elem : vec)
    {
        sum = sum + elem;
    }
    return sum / vec.size();
}

// Function to find standard
// deviation of given vector.
```

```
float standardDeviation(vector<float> vec)
{
    float sum = 0;
    for (auto& elem : vec)
    {
        sum = sum + (elem - Mean(vec)) *
            (elem - Mean(vec));
    }
    return sqrt(sum / (vec.size() - 1));
}

// Function to find t-test of
// two set of statistical data.
float tTest(vector<float> vec1, vector<float> vec2)
{
    float meanVal1 = Mean(vec1);
    float meanVal2 = Mean(vec2);
    float stdVal1 = standardDeviation(vec1);
    float stdVal2 = standardDeviation(vec2);

    // Formula to find t-test
    // of two set of data.
    float t_test = (meanVal1 - meanVal2) / sqrt((stdVal1 * stdVal2)
        / vec1.size() + (stdVal1 * stdVal2) / vec2.size());
    return t_test;
}

pair<vector<float>, vector<float>> getMeanandStdVectors(vector<vector<float>> v) {
    vector<float> meanVec;
    vector<float> stdVec;

    for (size_t i = 0; i < v.size(); ++i)
    {
        vector<float> myvector;
        for (size_t j = 0; j < v[i].size(); j++) {
            myvector.push_back(v[i][j]);
        }
        float meanVal = Mean(myvector);
        float stdVal = standardDeviation(myvector);
        meanVec.push_back(meanVal);
        stdVec.push_back(stdVal);
        myvector.clear();
    }

    return { meanVec, stdVec };
}

vector<float> getTtest(vector<vector<float>> disVec, vector<vector<float>> conVec) {
    vector<float> ttestVec;

    for (size_t i = 0; i < disVec.size(); ++i)
    {
        vector<float> diseaseVec, controlVec;
        for (size_t j = 0; j < disVec[i].size(); j++) {
            diseaseVec.push_back(disVec[i][j]);
        }
        for (size_t k = 0; k < conVec[i].size(); k++) {
            controlVec.push_back(conVec[i][k]);
        }
        float tTestVal = tTest(diseaseVec, controlVec);
        ttestVec.push_back(tTestVal);
        diseaseVec.clear();
        controlVec.clear();
    }

    return ttestVec;
}
```

```
// A function to randomly select
// k items from stream[0..n-1].
vector<float> returnKItems(vector<float> vec, int k)
{
    int i = 0;
    random_device rd;
    mt19937 g(rd());
    srand(time(NULL));
    shuffle(vec.begin(), vec.end(), g);
    vector<float> reservoir;
    for (auto& elem : vec) {
        if (i < k) {
            reservoir.push_back(elem);
        }
        i++;
        }
    return reservoir;
}

vector<float> replaceItems(vector<float> originalVec, vector<float> sampleVec1, vector<float> sampleVec2)
{
    for (int i=0; i < sampleVec1.size(); i++)
    {
        replace(originalVec.begin(), originalVec.end(), sampleVec1[i], sampleVec2[i]);
    }
    return originalVec;
}

vector<vector<float>> randomPermutation(vector<vector<float>> disVec, vector<vector<float>> conVec) {
    vector<vector<float>> randomTtestVec;
    randomTtestVec.resize(disVec.size());

    for (size_t i = 0; i < disVec.size(); ++i)
    {
        vector<float> diseaseVec, controlVec, sampleDisease, sampleControl;
        for (size_t j = 0; j < disVec[i].size(); j++) {
            diseaseVec.push_back(disVec[i][j]);
        }
        for (size_t k = 0; k < conVec[i].size(); k++) {
            controlVec.push_back(conVec[i][k]);
        }
        for (int m = 0; m < 1000; m++) {
            srand((unsigned)time(NULL) + m);
            sampleDisease = returnKItems(diseaseVec, 3);
            sampleControl = returnKItems(controlVec, 3);
            vector<float> newDiseaseVec = replaceItems(diseaseVec, sampleDisease, sampleControl);
            vector<float> newControlVec = replaceItems(controlVec, sampleControl, sampleDisease);
            float randomtTest = tTest(newDiseaseVec, newControlVec);
            randomTtestVec[i].push_back(randomtTest);
        }
        //cout << "Index: " << i << endl;
        //cout << "Random Permutation size: " << randomTtestVec[i].size() << endl;
        diseaseVec.clear();
        controlVec.clear();
    }
    return randomTtestVec;

}

vector<float> calculateDScore(vector<vector<float>> randPermVec, vector<float> ttestVec) {
    vector<float> dscoreVec, newVec;
    vector<string> newGeneVec, sortGenes;

    for (size_t i = 0; i < randPermVec.size(); ++i)
    {
        vector<float> randPerm;
        for (size_t j = 0; j < randPermVec[i].size(); j++) {
            randPerm.push_back(randPermVec[i][j]);
        }
        float meanRandPerm = Mean(randPerm);
        float stdRandPerm = standardDeviation(randPerm);
        float dScore = (ttestVec[i] - meanRandPerm) / stdRandPerm;
        dscoreVec.push_back(dScore);
        randPerm.clear();
    }



    return dscoreVec;
}

pair<vector<float>, vector<string>> sortDScore(vector<float> dscoreVec, vector<string> genVec) {
    vector<float> newVec;
    vector<string> sortGenes;

    //Sort in Descending order
    newVec.assign(dscoreVec.begin(), dscoreVec.end());
    vector<size_t> newVec2 = sort_indexes(newVec);
    reverse(newVec2.begin(), newVec2.end());
    newVec.clear();

    for (auto& elem : newVec2) {
        newVec.push_back(dscoreVec[elem]);
        sortGenes.push_back(genVec[elem]);
        cout << "D-Score: " << dscoreVec[elem] << "\tGene Index: " << genVec[elem] << endl;

    }
    return { newVec, sortGenes };
}



int main()
{
```

```
vector< vector<float > > matControl, matRenal, randomPermVecs;
vector< vector<string > > controlTwodVec, renalTwodVec, geneTwodVec, randPermStrVec;
vector<string> geneVec, sortGeneVec;
//matControl = parseCsvToVector("../control.csv");
renalTwodVec = parseCsvToVector("renal.csv");
controlTwodVec = parseCsvToVector("others.csv");
geneTwodVec = parseCsvToVector("gene_index.csv");
geneVec = flatten(geneTwodVec);
matRenal = return2DFloatVector(renalTwodVec);
matControl = return2DFloatVector(controlTwodVec);
auto tuples = returnVecwithKsize(matRenal, matControl, geneVec, 3); //remove gene with less than 3 patients
matRenal = get<0>(tuples);
matControl = get<1>(tuples);
geneVec = get<2>(tuples);
vector<float> meanRenal, stdRenal, meanControl, stdControl, ttstVec, dScoreVec;
auto renalVecs = getMeanandStdVectors(matRenal);
meanRenal = renalVecs.first;
stdRenal = renalVecs.second;
auto conVecs = getMeanandStdVectors(matControl);
meanControl = conVecs.first;
stdControl = conVecs.second;
for (size_t i = 0; i < meanRenal.size(); ++i)
{
   cout << "Gene Index: " << geneVec[i] << endl;
   cout << "mean Renal: " << meanRenal[i] << "\t mean Control: " << meanControl[i] << endl;
   cout << "Std Renal: " << stdRenal[i] << "\t std Control: " << stdControl[i] << endl;
}
auto start = std::chrono::high_resolution_clock::now();
ttstVec = getTtest(matRenal, matControl);
/*for (size_t i = 0; i < ttstVec.size(); ++i)
{
   cout << "Gene Index: " << geneVec[i] << endl;
   cout << "T test value: " << ttstVec[i] << endl;
}*/
```

```
    randomPermVecs = randomPermutation(matRenal, matControl);
    //randPermStrVec = return2DStringVector(randomPermVecs);
    //write_csv(randPermStrVec, "random_perm.csv");
    dScoreVec = calculateDScore(randomPermVecs, ttstVec);
    auto stop = std::chrono::high_resolution_clock::now();
    auto duration = std::chrono::duration_cast<std::chrono::seconds>(stop -start);
    printf("\nTotal Execution time %ld seconds\n", duration.count());
    auto sortVecs = sortDScore(dScoreVec, geneVec);
    dScoreVec = sortVecs.first;
    sortGeneVec = sortVecs.second;
    geneTwodVec.clear();
    geneTwodVec.resize(sortGeneVec.size());
    for (size_t i = 0; i < geneTwodVec.size(); ++i) {
        geneTwodVec[i].push_back(sortGeneVec[i]);
        geneTwodVec[i].push_back(to_string(dScoreVec[i]));
    }
    write_csv(geneTwodVec, "sort_gene_dscore.csv");

    return 0;

}
```

MPI Parallel Code:

```
#include <iostream>     // cout, endl
#include <fstream>      // fstream
#include <sstream>
#include <vector>
#include <string>
#include <algorithm>    // copy
#include <iterator>     // ostream_operator
#include <random>
#include <tuple>
#include <utility>
#include <cmath>
```

```
#include <numeric>      // std::iota
#include <algorithm>
#include <mpi.h>
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>

using namespace std;
#define SIZE 4532
#define renalSIZE 8
#define controlSIZE 52
#define randSize 5000

template<typename T>
std::vector<T> flatten(const std::vector<std::vector<T>>& orig)
{
    std::vector<T> ret;
    for (const auto& v : orig)
        ret.insert(ret.end(), v.begin(), v.end());
    return ret;
}

template <typename T>
vector<size_t> sort_indexes(const vector<T>& v) {

    // initialize original index locations
    vector<size_t> idx(v.size());
    iota(idx.begin(), idx.end(), 0);

    // sort indexes based on comparing values in v
    // using std::stable_sort instead of std::sort
    // to avoid unnecessary index re-orderings
    // when v contains elements of equal values
    stable_sort(idx.begin(), idx.end(),
        [&v](size_t i1, size_t i2) {return v[i1] < v[i2]; });

    return idx;
}

vector<vector<string>> parseCsvToVector(string fileName) {
    std::ifstream csv(fileName);
    std::string line;
    std::vector <std::vector<std::string>> matRows;

    if (csv.is_open()) {
        for (std::string row_line; std::getline(csv, row_line);)
        {
            matRows.emplace_back();
            std::istringstream row_stream(row_line);
            for (std::string column; std::getline(row_stream, column, ',');)
                matRows.back().push_back(column);

        }
        csv.close();
    }
    else {
        cout << "Unable to open file";
    }

    /*for (size_t i = 0; i < matRows.size(); i++) {
        for (size_t j = 0; j < matRows[i].size(); j++) {
            cout << matRows[i][j] << " ";
        }
        cout << endl;
    }*/
    int Nrows = matRows.size();
    int Ncols = matRows[0].size();
    cout << "number of rows: " << Nrows << endl;

    cout << "number of columns: " << Ncols << endl;

    return matRows;


}


void write_csv(vector<vector<string>> dataset, string filename) {


    // Create an output filestream object

    std::ofstream myFile(filename);


    // Send data to the stream

    for (int i = 0; i < dataset.size(); ++i)

    {

        for (int j = 0; j < dataset[i].size(); ++j)

        {

            myFile << dataset[i][j];

            if (j != dataset[i].size() - 1) myFile << ","; // No comma at end of line

        }

        myFile << "\n";

    }


    // Close the file

    myFile.close();

    cout << "Successfully written data into " << filename << endl;

}


vector<vector<float>> return2DFloatVector(vector<vector<string>> vec) {

    vector< vector<float > > matFloat;


    for (size_t i = 0; i < vec.size(); ++i)

    {

        matFloat.push_back(std::vector<float>());

        for (size_t j = 0; j < vec[i].size(); j++) {
            string s = vec[i][j];
            string chars = "";
            for (char c : chars) {
                s.erase(std::remove(s.begin(), s.end(), c), s.end());
            }
            if ((!s.empty()) & (s.size() > 1)) {
                matFloat[i].push_back(stof(s));
                //cout << matRows[i][j] << " ";
            }
        }
        //cout << endl;
    }

    int Nrows = matFloat.size();
    int Ncols = matFloat[0].size();
    cout << "number of rows: " << Nrows << endl;
    cout << "number of columns: " << Ncols << endl;
    return matFloat;

}

void return_2d_renal_array(vector<vector<float>> vec, float arr[SIZE][renalSIZE]) {

    //cout << "2D Float Arrays: " << endl;
    for (size_t i = 0; i < vec.size(); ++i)
        {
            //arr[i] = new float[vec[i].size()];
            for (size_t j = 0; j < vec[i].size(); j++) {
                arr[i][j] = vec[i][j];
                //cout << arr[i][j] << " ";
            }
            //arrFloat[i] = new float[vec[i].size()];

            //cout << endl;
        }

    /*int Nrows = sizeof (arr) / sizeof (arr[0]);
    int Ncols = sizeof (arr[0]) / sizeof (int);
    cout << "number of rows: " << Nrows << endl;
    cout << "number of columns: " << Ncols << endl;*/
}

void return_2d_ctrl_array(vector<vector<float>> vec, float arr[SIZE][controlSIZE]) {

    //cout << "2D Float Arrays: " << endl;
    for (size_t i = 0; i < vec.size(); ++i)
        {
            //arr[i] = new float[vec[i].size()];
            for (size_t j = 0; j < vec[i].size(); j++) {
                arr[i][j] = vec[i][j];
                //cout << arr[i][j] << " ";
            }
            //arrFloat[i] = new float[vec[i].size()];

            //cout << endl;
        }

    /*int Nrows = sizeof (arr) / sizeof (arr[0]);
    int Ncols = sizeof (arr[0]) / sizeof (int);
    cout << "number of rows: " << Nrows << endl;
    cout << "number of columns: " << Ncols << endl;*/
}


vector<float> return1DFloatVec(float arr[SIZE][1]) {
    vector<float> matFloat;

    for (size_t i = 0; i < SIZE; ++i)
    {
        matFloat.push_back(arr[i][0]);
    }

    int Nrows = matFloat.size();
    cout << "number of rows: " << Nrows << endl;
    return matFloat;

}

tuple<vector<vector<float>>,   vector<vector<float>>,   vector<string>>   returnVecwithKsize(vector<vector<float>>
disVec,
    vector<vector<float>> conVec, vector<string> geneIdVec, int k) {

    for (size_t i = 0; i < disVec.size(); ++i)
    {
        vector<float> myVec1, myVec2;
        for (size_t j = 0; j < disVec[i].size(); j++) {
            myVec1.push_back(disVec[i][j]);
        }
        for (size_t l = 0; l < conVec[i].size(); l++) {
            myVec2.push_back(conVec[i][l]);
        }
        if ((myVec1.size() < k) | (myVec2.size() < k)) {
            disVec.erase(disVec.begin()+i);
            conVec.erase(conVec.begin()+i);
            geneIdVec.erase(geneIdVec.begin()+i);
        }
        myVec1.clear();
        myVec2.clear();
    }

    cout << "number of Diesease rows: " << disVec.size() << endl;
    cout << "number of Control rows: " << conVec.size() << endl;
    cout << "number of Gene ID rows: " << geneIdVec.size() << endl;
    return { disVec, conVec, geneIdVec };

}

// Function to find mean.
float Mean(float arr[], int n)
{
    float sum = 0;
    for (int i = 0; i < n; i++)
        sum = sum + arr[i];
    return sum / n;
}

// Function to find standard
// deviation of given array.
float standardDeviation(float arr[], int n)
{
    float sum = 0;
    for (int i = 0; i < n; i++)
        sum = sum + (arr[i] - Mean(arr, n)) *
                    (arr[i] - Mean(arr, n));

    return sqrt(sum / (n - 1));
}

// Function to find t-test of
// two set of statistical data.
float tTest(float arr1[], int n,
            float arr2[], int m)
{
    float mean1 = Mean(arr1, n);
    float mean2 = Mean(arr2, m);
    float sd1 = standardDeviation(arr1, n);
    float sd2 = standardDeviation(arr2, m);

    // Formula to find t-test
    // of two set of data.
    float t_test = (mean1 - mean2) / sqrt((sd1 * sd1)
                            / n + (sd2 * sd2) / m);
    return t_test;
}


float *replaceItems(float *arr, float *oldArr, float *newArr)
{
    for (int i=0; i < sizeof (oldArr) / sizeof (oldArr[0]); i++)
    {
        replace(arr, arr + (sizeof (arr) / sizeof (arr[0])), oldArr[i], newArr[i]);
    }
    return arr;
}


int main(int argc, char **argv)
{
    int from, to;
    float renal2DArray [SIZE][renalSIZE];
    float control2DArray [SIZE][controlSIZE];
    float dScoreVec [SIZE][1];
    fill(renal2DArray[0], renal2DArray[0] + SIZE * renalSIZE, 0.000);
    fill(control2DArray[0], control2DArray[0] + SIZE * controlSIZE, 0.000);
    fill(dScoreVec[0], dScoreVec[0] + SIZE * 1, 0.000);
    vector< vector<float > > matControl, matRenal;
    vector< vector<string > > controlTwodVec, renalTwodVec, geneTwodVec, randPermStrVec;
    vector<string> geneVec, sortGeneVec;
    vector<float>ttstVec, dScores, newVec;
    renalTwodVec = parseCsvToVector("renal.csv");
```

```
controlTwodVec = parseCsvToVector("others.csv");
geneTwodVec = parseCsvToVector("gene_index.csv");
geneVec = flatten(geneTwodVec);
matRenal = return2DFloatVector(renalTwodVec);
matControl = return2DFloatVector(controlTwodVec);
auto tuples = returnVecwithKsize(matRenal, matControl, geneVec, 3); //remove gene with less than 3 patients
matRenal = get<0>(tuples);
matControl = get<1>(tuples);
geneVec = get<2>(tuples);
return_2d_renal_array(matRenal, renal2DArray);
return_2d_ctrl_array(matControl, control2DArray);

/*for (size_t i = 0; i < SIZE; ++i)
{
    //Ncols = matControl[i].size();
    cout << i << " <--Index NUmber"<<  endl;
    int m = 0;
    //cout << Ncols << " <--num of columns"<<  endl;
    for (size_t j = 0; j < controlSIZE; j++) {
        if (abs(control2DArray[i][j]) > 0.000) {
            cout << control2DArray[i][j] << " ";
            m++;
        }
        //cout << control2DArray[i][j] << " ";
    }



    cout << "\n Column Size: " << m << endl;
}*/

MPI_Init(NULL, NULL);
int world_rank;
MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
int world_size;
```

```
MPI_Comm_size(MPI_COMM_WORLD, &world_size);

from = world_rank * SIZE/world_size;
to = ((world_rank+1) * SIZE/world_size);

/*if(world_rank==0){

}*/
double start_time = MPI_Wtime();
MPI_Bcast (renal2DArray, SIZE*renalSIZE, MPI_INT, 0, MPI_COMM_WORLD);
//printf("\n\n");
if(world_rank==0){
    MPI_Scatter (&control2DArray[0][0], SIZE*controlSIZE/world_size, MPI_INT, MPI_IN_PLACE,
SIZE*controlSIZE/world_size, MPI_INT, 0, MPI_COMM_WORLD);
}
else{
    MPI_Scatter (&control2DArray[0][0], SIZE*controlSIZE/world_size, MPI_INT, &control2DArray[from][0],
SIZE*controlSIZE/world_size, MPI_INT, 0, MPI_COMM_WORLD);
}
MPI_Barrier(MPI_COMM_WORLD);
printf("computing processors %d (from row %d to %d)\n",world_rank, from, to-1);
for (size_t i=from; i<to; i++) {
    int ren = 0;
    float subRenal[renalSIZE];
    for (size_t k=0; k<renalSIZE; k++){
        if (abs(renal2DArray[i][k]) > 0.000) {
            subRenal[ren] = renal2DArray[i][k];
            ren++;
        }
    }
    int ctr = 0;
    float subControl[controlSIZE];
    for (size_t v=0; v<controlSIZE; v++){
        if (abs(control2DArray[i][v]) > 0.000) {
            subControl[ctr] = control2DArray[i][v];
            ctr++;
        }
    }
    float tTst= tTest(subRenal, ren, subControl, ctr);
    float randomPerms[randSize];
    for (int n = 0; n < randSize; n++) {
        srand((unsigned)time(NULL) + world_rank*world_size + n);
        random_device rd;
        mt19937 g(rd());
        shuffle(subRenal, subRenal + ren, g);
        shuffle(subControl, subControl + ctr, g);
        float renalRand[3] = {subRenal[0], subRenal[1], subRenal[2]};
        float ctrlRand[3] = {subControl[0], subControl[1], subControl[2]};
        float *newRenal = replaceItems(subRenal, renalRand, ctrlRand);
        float *newCtrl = replaceItems(subControl, ctrlRand, renalRand);
        randomPerms[n] = tTest(newRenal, ren, newCtrl, ctr);
    }
    dScoreVec[i][0] = (tTst - Mean(randomPerms, randSize))/standardDeviation(randomPerms, randSize);
    fill(subRenal, subRenal+renalSIZE, 0);
    fill(subControl, subControl+controlSIZE, 0);
    fill(randomPerms, randomPerms+randSize, 0);

}

if(world_rank==0){
   MPI_Gather (MPI_IN_PLACE, SIZE/world_size, MPI_INT, &dScoreVec[0][0], SIZE/world_size, MPI_INT, 0,
MPI_COMM_WORLD);
}else{
   MPI_Gather (&dScoreVec[from][0], SIZE/world_size, MPI_INT, &dScoreVec[0][0], SIZE/world_size, MPI_INT, 0,
MPI_COMM_WORLD);
}
double end_time = MPI_Wtime();

if (world_rank==0) {
    //sort in descending order
    dScores = return1DFloatVec(dScoreVec);

    newVec.assign(dScores.begin(), dScores.end());

    vector<size_t> newVec2 = sort_indexes(newVec);

    reverse(newVec2.begin(), newVec2.end());

    newVec.clear();

    geneTwodVec.clear();

    geneTwodVec.resize(SIZE);

    printf("\n\n");

    int i = 0;

    {

        for (auto& elem : newVec2) {

            cout << "Gene Index: " << geneVec[elem] << endl;

            printf("D score: %f ", dScores[elem]);

            printf("\n");

            geneTwodVec[i].push_back(geneVec[elem]);

            geneTwodVec[i].push_back(to_string(dScores[elem]));

            i++;

        }

        printf("Total Execution time: %f seconds \n", end_time - start_time);

        write_csv(geneTwodVec, "sort_gene_dscore_MPI.csv");

    }


    printf("\n\n");


}


MPI_Finalize();

return 0;


}
```