MPI-based Genome Wide Association Study

CIS667 - Julian Carrasquillo - 11/21/19

Specification

The basic idea is to develop an MPI-based message-passing application that analyzes raw microarray data to identify differences in gene expression between two user-defined groups of patient samples.

Background

The ongoing development of cDNA microarray technology has facilitated the simultaneous measurement and comparison of gene expression on a genomic scale. One technique uses the fluorescent intensity ratios of differentially expressed genes to characterize gene expression patterns. These patterns, in turn, can reveal the gene sets that underlie a particular phenotype or that represent a regulatory gene defect. For example, a molecular profiling study might attempt identification of gene expression signatures to distinguish groups of samples based on various specified parameters (e.g. differentiation state, tumor stage, tumor grade).

Discrimination Algorithm

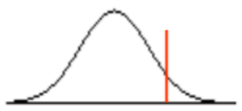
To determine expression signatures, individual genes are scored and ranked based on an appropriate discrimination metric. The algorithm first groups the individual gene expression values based on the supplied sample labeling; for example, diseased vs. normal individuals. Then the Student's t-statistic is calculated and used to identify significant discriminators (i.e. genes that significantly distinguish between the two sample groups). An effective form of the t-statistic is:

$$\frac{\bar{x}_1 - \bar{x}_2}{\sqrt{\frac{\sigma_1^2}{n_1} + \frac{\sigma_2^2}{n_2}}}$$

where n_1 and n_2 are the sizes of the two groups, $\bar{x}_1$ and $\bar{x}_2$ are the means of the two groups, and σ_1 and σ_2 are the sample standard deviations of the two groups. The resulting t-stat value may be either positive or negative; the magnitude of that value represents how "different" the two sample groups are when that discriminator is used.

The significance of the resulting t-statistic can be determined via **bootstrap** analysis; that is, by comparison to a distribution of t-statistics generated by simulating new samples by building new sets of the same size by selecting random records *with replacement* from the original.

For each gene a distribution of randomly generated t-statistics is created. This distribution has a mean and a 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 (see diagram below left). If the initial t-statistic (the red line) is ≥ 3 standard deviations from the mean of the distribution formed using random sample permutations, then with 97% confidence that gene is a discriminator.



$$D = \frac{|t_S - \mu_D|}{\sigma_D}$$

Each gene is ranked by its discrimination score (D, above right), where t_S is the tstatistic of the reference sample, μ_d is 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.

Microarray Data

This project uses the anonymized microarray patient data from the National Cancer Institute (NCI-60) in an attempt to identify genes that best serve as discriminants for renal cancer ("RE"). There are eight diseased samples, out of 60 total patients (the columns). There are 4549 total genes (the rows).

Note: an empty cell indicates that data does not exist for that gene for that patient; you must adjust your statistical calculations accordingly to account for the difference(s) in group size.

Note 2: some pre-processing has been performed on the dataset:

- it is in the Comma-Separated Values (.csv) format
- it only includes Gene ID and intensity values
- Gene 4538X (disease sample size of 1) has been removed from the dataset

Methodology

Reading In

The `csv` file was read in using a vector string. This allowed for maintenance of the gene name along with each of the patient measurements.

Splitting the Work

The split of the work occurred along the rows. This allowed for easier indexing and keeping related records together. Based on the number of nodes incorporated into the program, the gene table was split into chunks of relatively equal sizes. Using integer division, the number of rows assigned to each node was a whole number, with the final node of the group being assigned the remainder along with its normal share. This added a hurdle that will be discussed later in this write up.

```
start_interval = 0.0;
end_interval = 4549;
groups = end_interval / num_nodes;
remaining_rows = end_interval % num_nodes;

a = start_interval + my_rank*groups;
b = a + groups - 1;

// tack on remaining rows to the last node
if(my_rank == num_nodes - 1) b = b + remaining_rows;
```

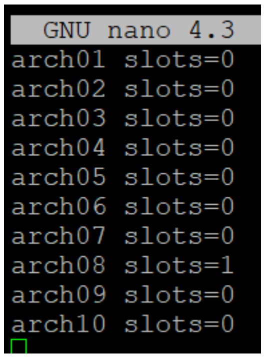
MPI Implementation

Each node created a local array to fill with d scores, the size of which was made using their unique a and b values. At each row, the program loops through each patient and extracts their value. This array was then sent to the `make_ttest()` function, which handled the missing values. `make_ttest` returns an object of class `sample_stats`. It includes the original array of values, the calculated t stat, and the counts of non-empty test and control patients. The object was passed to `bootstrapper()` which generates random indices, builds a new sample array, then passes it back to `make_ttest()` to generate a t stat. Running this 500 times yielded a bootstrapped distribution. `MPI_Gather` was leveraged for communications. This freed me from having to implement conditionals and avoid the indexing issues that can arise in those scenarios.

```
double local_dscores[b - a];

for(int i = a; i <= b; i++){
  // skip header
    if(i == 0) {
        continue;
    }
    for(int j = 0; j < 61; j++){
        sample[j] = row[i*61 + j];
    }

    key_stats = make_ttest(sample);
    for(int j = 0; j < 500; j++){
        bootstrapped_distro[j] = bootstrapper(key_stats);
    }
    local_dscores[i - a] = calc_D_score(bootstrapped_distro, key_stats.t_stat, 500);

}
MPI_Gather(local_dscores,
  remaining_rows + b - a, MPI_DOUBLE, d_scores,
  remaining_rows + b - a, MPI_DOUBLE, MASTER, MPI_COMM_WORLD);
```

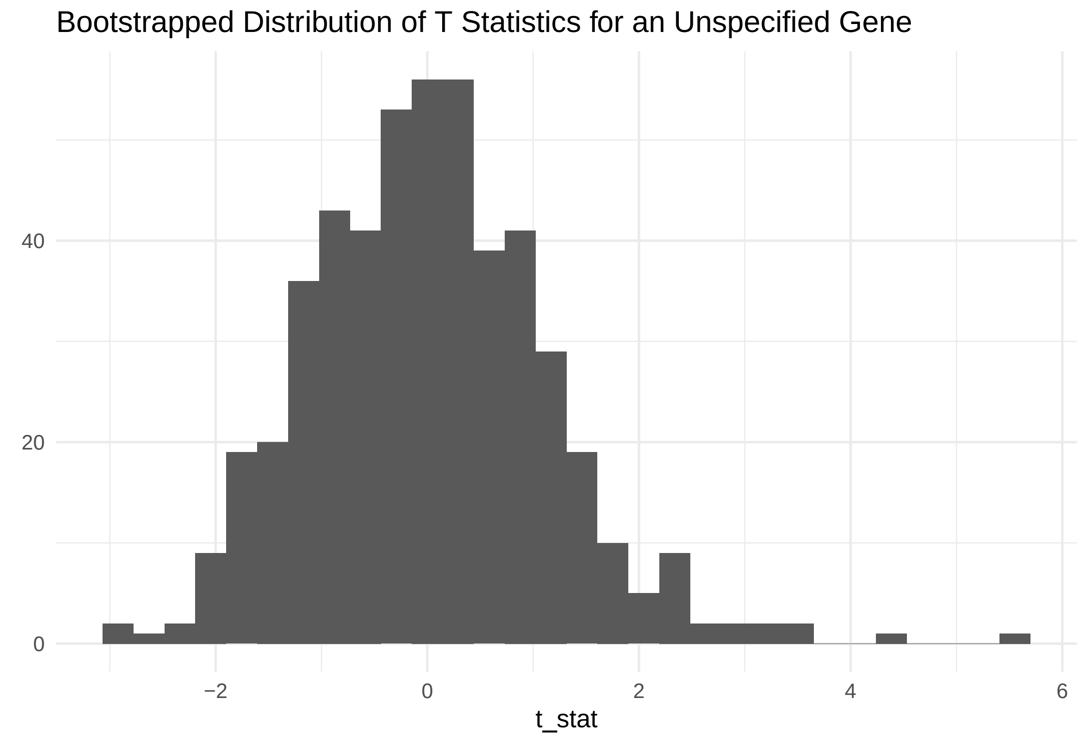
The `b - a` size initially kicked out segmentation fault errors. This was because even though size `b - a` was enough *most of the time*, there was always going to be one node that had more values to return because of the `remaining_rows` assigned. To remedy, the worse-case scenario size of `remainding_rows + b - a` was used to handle all incoming streams.

Timing and outputs logs were written to `csv` for analysis in R. The sequential benchmark was found by setting the `my_hosts` file to a single node within the local machine. In these experiments, `arch08` was the local machine.



```
  GNU nano 4.3
arch01 slots=0
arch02 slots=0
arch03 slots=0
arch04 slots=0
arch05 slots=0
arch06 slots=0
arch07 slots=0
arch08 slots=1
arch09 slots=0
arch10 slots=0
```

See below distribution of t statistics for one of the genes. In this scenario, the bootstrapping implies that the difference in gene expression between the test and control groups is near 0.



Results

A Few Outputs

Below shows 2 6-node runs as well as the single node run which was used for timed benchmarking.

Table 1: 6 nodes

Gene ID	D-Score	|D-Score|
GENE3941X	7.55581	7.55581
GENE2107X	7.38214	7.38214
GENE377X	7.11458	7.11458
GENE2820X	7.03358	7.03358
GENE3554X	6.98652	6.98652
GENE3635X	6.80138	6.80138
GENE3553X	6.74746	6.74746
GENE4364X	6.65832	6.65832
GENE4007X	6.63293	6.63293
GENE2335X	6.41471	6.41471

Table 2: 1 node / Sequential

Gene ID	D-Score	|D-Score|
GENE3635X	7.36646	7.36646
GENE364X	-7.16788	7.16788
GENE3554X	7.07648	7.07648
GENE3941X	6.85938	6.85938
GENE4364X	6.71023	6.71023
GENE377X	6.50781	6.50781
GENE3886X	6.40343	6.40343
GENE2107X	6.40290	6.40290
GENE2833X	6.23801	6.23801
GENE3553X	6.11057	6.11057

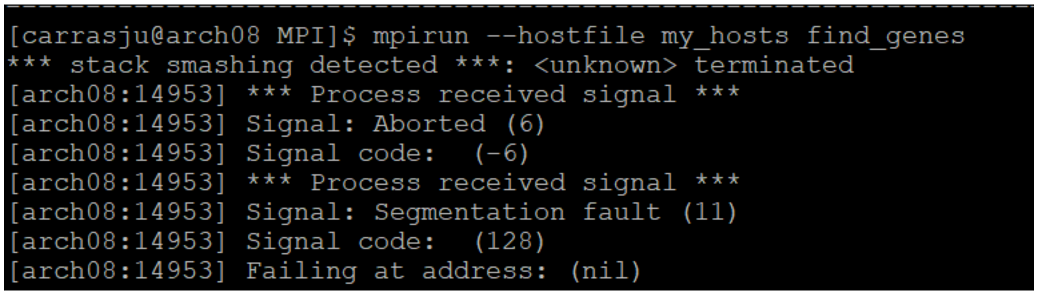
Table 3: 6 nodes

Gene ID	D-Score	|D-Score|
GENE377X	8.28326	8.28326
GENE3554X	7.91919	7.91919
GENE364X	-7.36761	7.36761
GENE2107X	6.95811	6.95811
GENE3635X	6.92416	6.92416
GENE2891X	6.83634	6.83634
GENE4364X	6.28384	6.28384
GENE4007X	6.24921	6.24921
GENE3714X	6.01280	6.01280
GENE2820X	5.97109	5.97109

The top ten genes in these tables show D-Scores of 6 to 8. With a score of 3 being enough to satisfy a 3% significance level, these expression levels in the test group are quite different from the control. There are consistencies across the 3 tables, but the differences show the nature of us introducing a random element to the analysis.

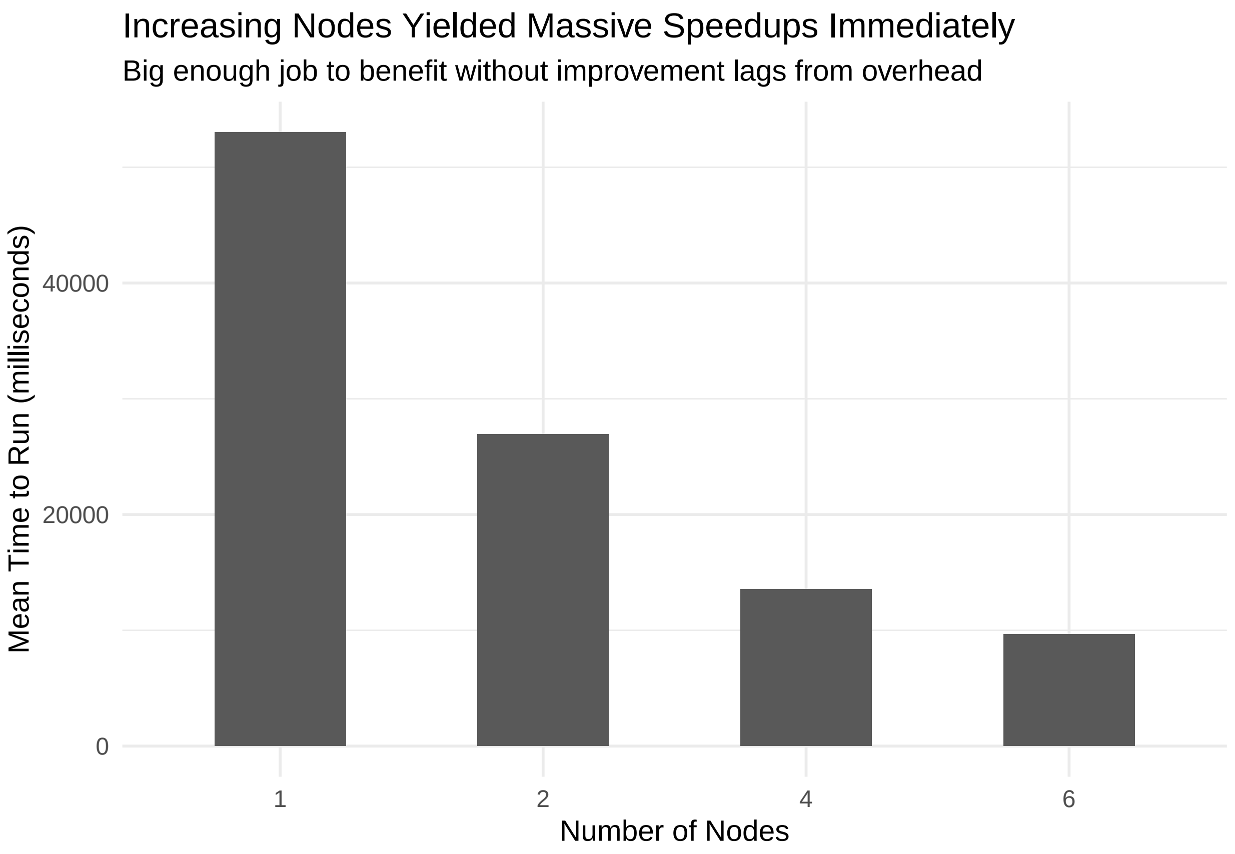
Speed Up Analysis

I was able to increase the number of nodes in the program to 6, with anything more than that generating a `stack smashing detection` error:



```
------------------------------------------------------------------------
[carrasju@arch08 MPI]$ mpirun --hostfile my_hosts find_genes
*** stack smashing detected ***: <unknown> terminated
[arch08:14953] *** Process received signal ***
[arch08:14953] Signal: Aborted (6)
[arch08:14953] Signal code:  (-6)
[arch08:14953] *** Process received signal ***
[arch08:14953] Signal: Segmentation fault (11)
[arch08:14953] Signal code:  (128)
[arch08:14953] Failing at address: (nil)
```

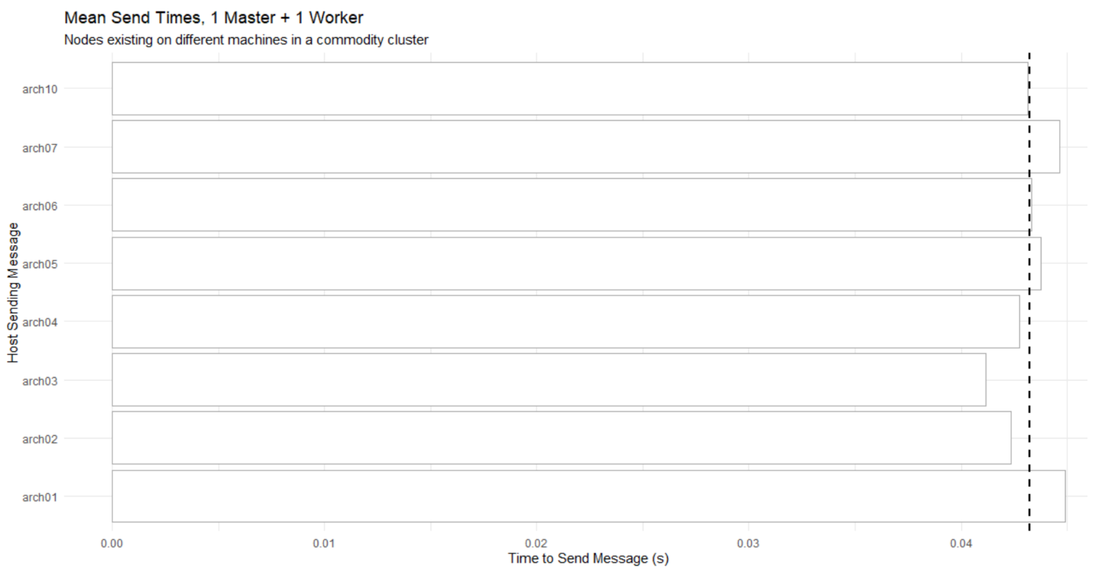
Potentials issues could be coming from the `MPI_Gather` call, but I did not push too much because there were realized speed ups from 1 to 6 nodes. Adding nodes showed almost linear speedup



# of Nodes	Mean Run Time (ms)	Observed Speedup
1	53011.667	1.000000
2	26930.400	1.968469
4	13545.000	3.913744
6	9662.429	5.486371

Revisiting Pairs of Machines

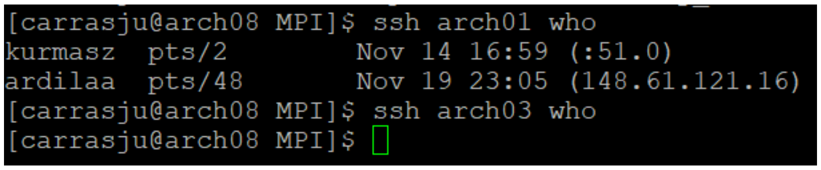
Last week, I looked into the potential differences in communication speeds between pairs of machines in the arch cluster. Knowing which nodes in the cluster speak better with each other is a potentially $100M answer[1]. In that analysis, there was some possible evidence suggesting that `arch08` had better messaging speed with `arch03` against the average of the whole cluster, while `arch08` and `arch01` had less strong results.



To experiment, I looked at a 6-node set up with 3 on the home `arch08` and the other 3 on either `arch01` or `arch03`. Using 10 runs for each set up, it looks like `arch03` is still showing better than `arch01`.

Machine	Mean Time to Run, in ms
arch01	33701.200
arch03	9203.444

Of course, I'd have to disclose that there was a difference in user counts in both. This is definitely worth exploring more.



```
[carrasju@arch08 MPI]$ ssh arch01 who
kurmasz  pts/2        Nov 14 16:59 (:51.0)
ardilaa  pts/48       Nov 19 23:05 (148.61.121.16)
[carrasju@arch08 MPI]$ ssh arch03 who
[carrasju@arch08 MPI]$
```

Conclusions

It was a pleasant surprise to have HPC meet my statistical background. A lot can be done with statistical simulation on a cheaper scale versus running actual experiment. With teaching applications, we can see how nature acts with a known distribution and apply that to the natural world. Microarrays are fascinating and seem to be a cornerstone of biological studies.

MPI has been my favorite HPC tool in terms of ease of use and experimentation. It is easy to grasp the core concept and implement simple programs using the 6 main functions. That said, there is plenty of opportunity to really get into the weeds with some of the more advance topics.

Next steps would definitely include finding why the program errors out after 6 nodes. It could be an issue with the dynamic array building using the end points of each nodes' work split.

[1] https://www.cnbc.com/2019/08/01/goldman-spending-100-million-to-shave-milliseconds-off-stock-trades.html

Apendix

`class sample_stats`

```
class sample_stats
{
  public:
    double t_stat;
    int non_empty_test;
        int non_empty_ctrl;
        double converted[60];
};
```

make_ttest calculated means and sample standard deviations

```
sample_stats make_ttest(string sample_string[]){
    double converted[60];
    int non_empty_test = 0;
    int non_empty_ctrl = 0;
    double sum_test = 0;
    double sum_ctrl = 0;
    double mean_test, sd_sq_test, mean_ctrl, sd_sq_ctrl, t_stat;
    double temp = 0;
    sample_stats key_stats;

    for(int i = 1; i <= 60; i++){
        converted[i - 1] = atof(sample_string[i].c_str());
        // add them to the returned class
        key_stats.converted[i - 1] = converted[i - 1];
    }

    // get the mean for all non 0 items
    for(int i = 0; i < 8; i++){
        if(converted[i] != 0){
            sum_test = sum_test + converted[i];
            non_empty_test++;
        }
    }
    mean_test = sum_test / non_empty_test;

    //calculate sd now that we have the mean
    for(int i = 0; i < 8; i++){
        if(converted[i] != 0){
            temp = temp + pow(converted[i] - mean_test, 2);
        }
    }
    sd_sq_test = temp / (non_empty_test - 1);

    temp = 0;
    for(int i = 8; i < 60; i++){
        if(converted[i] != 0){
            sum_ctrl = sum_ctrl + converted[i];
            non_empty_ctrl++;
        }
    }
    mean_ctrl = sum_ctrl / non_empty_ctrl;

    //calculate sd now that we have the mean
    for(int i = 8; i < 60; i++){
        if(converted[i] != 0){
            temp = temp + pow(converted[i] - mean_ctrl, 2);
        }
    }
    sd_sq_ctrl = temp / (non_empty_ctrl - 1);

    t_stat = (mean_test - mean_ctrl)/(sqrt((sd_sq_test/non_empty_test) + (sd_sq_ctrl/non_empty_ctrl)));
    key_stats.t_stat = t_stat;
    key_stats.non_empty_ctrl = non_empty_ctrl;
    key_stats.non_empty_test = non_empty_test;
    return key_stats;
}
```

`bootstrapper` rebuilt a dataset and fed it back into `make_ttest`

```
double bootstrapper(sample_stats key_stats){
    double bootstrapped[60];
    string bootstrapped_str[61];
    int temp;
    int test_count = 0;
    int ctrl_count = 0;
    bool index_not_empty = false;
    sample_stats bootstrapped_stats;

    // load bootstrapped with the bootstrapped index
    for(int i = 0; i < 60; i++){
        index_not_empty = false;
        // get a random index - check to see if it is empty. If so, get a new number
        while(!index_not_empty){
            temp = rand() % 60;
            if(key_stats.converted[temp] != 0){
                index_not_empty = true;
            }
        }
        // if we're in the test zone (first 8 spots) AND we have not reached
        // the number of non empty test entries, place the value.
        // If not zero out and let make_ttest() handle later.
        if(i < 8 & test_count <= key_stats.non_empty_test){
            bootstrapped[i] = key_stats.converted[temp];
            test_count++;
        } else if(i < 8 & i > key_stats.non_empty_test) {
            bootstrapped[i] = 0;
        } else if(i < 60 & ctrl_count <= key_stats.non_empty_ctrl){
            bootstrapped[i] = key_stats.converted[temp];
            ctrl_count++;
        } else {
            bootstrapped[i] = 0;
        }

        bootstrapped_str[i + 1] = to_string(bootstrapped[i]);
    }
    bootstrapped_stats = make_ttest(bootstrapped_str);
    return bootstrapped_stats.t_stat;
}
```

D-score calculator

```
double calc_D_score(double bootstraps[], double t_stat, int bootstrap_size){
    double sum = 0;
    double bootstrap_mean, bootstrap_sd, temp, d_score;

    for(int i = 0; i < bootstrap_size; i++){
        sum += bootstraps[i];
    }
    bootstrap_mean = sum / bootstrap_size;

    //calculate sd now that we have the mean
    for(int i = 0; i < bootstrap_size; i++){
        temp += pow(bootstraps[i] - bootstrap_mean, 2);
    }
    bootstrap_sd = temp / bootstrap_size;
    d_score = (t_stat - bootstrap_mean) / bootstrap_sd;
    return d_score;
}
```

Main

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <mpi.h>
#include <fstream>
#include <vector>
#include <sstream>
#include <math.h>
#include <chrono>

#define MASTER  0
#define TAG     0
using namespace std;

int main(int argc, char* argv[])
{
  int my_rank, source, num_nodes, groups, remaining_rows, end_interval, a, b;
    char my_host[MAX];
    double start_time, end_time, start_interval;
    double bootstrapped_distro[500];
    double d_scores[4549];
    string sample[61];
    sample_stats key_stats;
    ofstream logFile;

    // make sure randomly generated bootstraps are different each run
    srand(time(NULL));

    // Read in file
    fstream file_in;
    file_in.open("NCI-60.csv", ios::in);
    vector<string> row;
    string line, word;

    while(getline(file_in, line)){
        // used for breaking words
        stringstream s(line);

        // read every column data of a row and
        // store it in a string variable, 'word'
        while (getline(s, word, ',')) {
            // add all the column data
            // of a row to a vector
            row.push_back(word);
        }
    }
    file_in.close();

    auto start = chrono::steady_clock::now();
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &num_nodes);

    start_interval = 0.0;
    end_interval = 4549;
    groups = end_interval / num_nodes;
    remaining_rows = end_interval % num_nodes;

    a = start_interval + my_rank*groups;
    b = a + groups - 1;

    // tack on remaining rows to the last node
    if(my_rank == num_nodes - 1) b = b + remaining_rows;

    double local_dscores[b - a];

    for(int i = a; i <= b; i++){
        // skip header
        if(i == 0) {
            continue;
        }
        for(int j = 0; j < 61; j++){
            sample[j] = row[i*61 + j];
        }

        key_stats = make_ttest(sample);
        for(int j = 0; j < 500; j++){
            bootstrapped_distro[j] = bootstrapper(key_stats);
        }
        local_dscores[i - a] = calc_D_score(bootstrapped_distro, key_stats.t_stat, 500);

    }
    MPI_Gather(local_dscores,
      remaining_rows + b - a, MPI_DOUBLE, d_scores,
      remaining_rows + b - a, MPI_DOUBLE, MASTER, MPI_COMM_WORLD);

    auto end = chrono::steady_clock::now();

    logFile.open("logging.txt", ios_base::app);
    logFile << num_nodes << ","
        << chrono::duration_cast<chrono::milliseconds>(end - start).count() << ","
        << "ms" << endl;
    logFile.close();

    logFile.open("d_scores.txt", ios_base::app);
    for(int i = 1; i <= 4549; i++){
        logFile << d_scores[i] << endl;
    }
    logFile.close();

    MPI_Finalize();

    return 0;
}
```