CED19I028
Sumit Kumar

MPI Parallelization

About Problem Statement

Genetic Algorithms (GAs) are adaptive heuristic search algorithms that belong to the larger part of evolutionary algorithms. Genetic algorithms are based on the ideas of natural selection and genetics.
These are intelligent exploitation of random search provided with historical data to direct the search into the region of better performance in solution space.
Genetic algorithms simulate the process of natural selection which means those species who can adapt to changes in their environment are able to survive and reproduce and go to next generation. In simple words, they simulate “survival of the fittest” among individual of consecutive generation for solving a problem. Each generation consist of a population of individuals and each individual represents a point in search space and possible solution. Each individual is represented as a class object. In class object there is string which is analogous to the Chromosome.

Reference - https://www.geeksforgeeks.org/genetic-algorithms/

Serial C++ Code

```
%%cu

// C++ program to create target string, starting from random string using Genetic Algorithm
#include "iostream"
#include "vector"
#include "time.h"
#include "algorithm"
using namespace std;
// Number of individuals in each generation
#define POPULATION_SIZE 1000
// #define POPULATION_SIZE 500000
int generation = 0;
// Valid Genes
const string GENES =
"`~1!2@3#4$5%6^7&8*9(0)-_+QWERTYUIOPASDFGHJKLZXCVBNMqwertyuiopasdfghjklzxcvbnm[{]}|;:'\",./?
< >";
```

```
// Target string to be generated
const string TARGET = "Random Generation...";

// Create random genes for mutation
char mutated_genes() {
   int len = GENES.size();
   return GENES[rand()%len];
}
// create chromosome or string of genes
string create_gnome() {
   int len = TARGET.size();
   string gnome = "";
   for(int i=0; i<len; i++)
       gnome += mutated_genes();
   return gnome;
}
// Class representing individual in population
class Individual {

   public:
       string chromosome;
       int fitness;

       Individual(string chromosome);
};
Individual::Individual(string chromosome) {
   this->chromosome = chromosome;

   int len = TARGET.size();
   int offspring_fitness = 0;
   // #pragma omp parallel for shared(TARGET, chromosome) reduction(+:fitness)
   for(int i=0; i<len; i++) {
       if(chromosome[i] != TARGET[i]){
           offspring_fitness++;
       }
   }
   this->fitness = offspring_fitness;
};
// Overloading < operator
bool compare(Individual* ind1, Individual* ind2) {
   return ind1->fitness < ind2->fitness;
}
// Driver code
int main() {

   double start_time, end_time;
   int threads;
   srand(time(0));
   vector<Individual*> population(POPULATION_SIZE);
   bool found = false;
   // create initial population
```

```
    for(int i=0; i<POPULATION_SIZE; i++) {
        string gnome = create_gnome();
        population[i] = new Individual(gnome);
    }

    while(!found) {

        // sort the population in increasing order of fitness score
        sort(population.begin(), population.end(), compare);

        // if the individual having lowest fitness score ie. 0 then we know that we have
reached to the target and break the loop
        if(population[0]->fitness <= 0){
            found = true;
            break;
        }

        // Otherwise generate new offsprings for new generation
        vector<Individual*> new_generation(POPULATION_SIZE);

        // Perform Elitism, that mean 10% of fittest population goes to the next generation
        int s = (10*POPULATION_SIZE)/100;

        for(int i=0; i<s; i++){
            new_generation[i] = population[i];
        }

        // From 50% of fittest population, Individuals will mate to produce offspring
        int right = (50*POPULATION_SIZE)/100;

        for(int i=s; i<POPULATION_SIZE; i++) {

            // int r = random_num(0, right);
            int r = rand()%(right+1);
            Individual* parent1 = population[r];

            // r = random_num(0, right);
            r = rand()%(right+1);
            Individual* parent2 = population[r];

            // chromosome for offspring
            string child_chromosome = "";
            string chromosome = parent1->chromosome;
            int len = chromosome.size();

            for(int i = 0;i<len;i++) {
                // random probability
                float p = (rand()%101)/100;

                // if prob is less than 0.45, insert gene from parent 1
                if(p < 0.45)
                    child_chromosome += chromosome[i];
```

```
            // if prob is between 0.45 and 0.90, insertgene from parent 2
            else if(p < 0.90)
                child_chromosome += parent2->chromosome[i];
            // otherwise insert random gene(mutate), for maintaining diversity
            else
                child_chromosome += mutated_genes();
        }

        // create new Individual(offspring) using generated chromosome for offspring
        Individual* offspring = new Individual(child_chromosome);

        new_generation[i] = offspring;
    }

    population = new_generation;
    cout<< "Generation: " << generation << "\t";
    cout<< "String: "<< population[0]->chromosome <<"\t";
    cout<< "Fitness: "<< population[0]->fitness << "\n";

    generation++;

  }

  cout<< "Generation: " << generation << "\t";
  cout<< "String: "<< population[0]->chromosome <<"\t";
  cout<< "Fitness: "<< population[0]->fitness << "\n";

 return 0;
}
```

Flat Profile of Serial Code

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 15.35      1.32     1.32 12000000     0.00     0.00  Individual::cal_fitness()
 10.23      2.20     0.88 11000000     0.00     0.00  Individual::mate(Individual)
  9.48      3.02     0.81 373159911     0.00     0.00  random_num(int, int)
  5.70      3.50     0.49                             _init
  5.29      3.96     0.46 725254058     0.00     0.00  __gnu_cxx::__normal_iterator<Individual*, std::vector<Individual, std::allocator<Individual> > >::operator*() const
  4.13      4.32     0.35 246659903     0.00     0.00  operator<(Individual const&, Individual const&)
  4.13      4.67     0.35 108166060     0.00     0.00  std::enable_if<std::__and_<std::__not_<std::__is_tuple_like<Individual> >, std::is_move_constructible<Individual>, std::is_move_assigna
ble<Individual> >::value, void>::type std::swap<Individual>(Individual&, Individual&)
  3.78      5.00     0.33 232457779     0.00     0.00  bool __gnu_cxx::__ops::_Iter_less_iter::operator()<__gnu_cxx::__normal_iterator<Individual*, std::vector<Individual, std::allocator<Ind
ividual> > >, __gnu_cxx::__normal_iterator<Individual*, std::vector<Individual, std::allocator<Individual> > > >(__gnu_cxx::__normal_iterator<Individual*, std::vector<Individual, std::alloca
tor<Individual> > >, __gnu_cxx::__normal_iterator<Individual*, std::vector<Individual, std::allocator<Individual> > >) const
  3.49      5.29     0.30  1284267     0.00     0.00  __gnu_cxx::__normal_iterator<Individual*, std::vector<Individual, std::allocator<Individual> > > std::__unguarded_partition<__gnu_cxx::_
_normal_iterator<Individual*, std::vector<Individual, std::allocator<Individual> > >, __gnu_cxx::__ops::_Iter_less_iter>(__gnu_cxx::__normal_iterator<Individual*, std::vector<Individual, std
::allocator<Individual> > >, __gnu_cxx::__normal_iterator<Individual*, std::vector<Individual, std::allocator<Individual> > >, __gnu_cxx::__normal_iterator<Individual*, std::vector<Individua
l, std::allocator<Individual> > >, __gnu_cxx::__ops::_Iter_less_iter)
  3.37      5.58     0.29 230534256     0.00     0.00  Individual::operator=(Individual&&)
  2.85      5.83     0.24 147483262     0.00     0.00  Individual::Individual(Individual&&)
  2.79      6.07     0.24 378017518     0.00     0.00  std::remove_reference<Individual&>::type&& std::move<Individual&>(Individual&)
  2.56      6.29     0.22 205683262     0.00     0.00  Individual::~Individual()
  2.21      6.48     0.19                             void std::__partial_sort<__gnu_cxx::__normal_iterator<Individual*, std::vector<Individual, std::allocator<Individual> > >, __gnu_cxx::__
ops::_Iter_less_iter>(__gnu_cxx::__normal_iterator<Individual*, std::vector<Individual, std::allocator<Individual> > >, __gnu_cxx::__normal_iterator<Individual*, std::vector<Individual, std:
:allocator<Individual> > >, __gnu_cxx::__normal_iterator<Individual*, std::vector<Individual, std::allocator<Individual> > >, __gnu_cxx::__ops::_Iter_less_iter)
  1.98      6.65     0.17 108166060     0.00     0.00  void std::iter_swap<__gnu_cxx::__normal_iterator<Individual*, std::vector<Individual, std::allocator<Individual> > >, __gnu_cxx::__norm
al_iterator<Individual*, std::vector<Individual, std::allocator<Individual> > > >(__gnu_cxx::__normal_iterator<Individual*, std::vector<Individual, std::allocator<Individual> > >, __gnu_cxx:
:__normal_iterator<Individual*, std::vector<Individual, std::allocator<Individual> > >)
  1.74      6.80     0.15 108166060     0.00     0.00  bool __gnu_cxx::operator< <Individual*, std::vector<Individual, std::allocator<Individual> > >(__gnu_cxx::__normal_iterator<Individual*
, std::vector<Individual, std::allocator<Individual> > > const&, __gnu_cxx::__normal_iterator<Individual*, std::vector<Individual, std::allocator<Individual> > > const&)
  1.51      6.93     0.13 130737308     0.00     0.00  __gnu_cxx::__normal_iterator<Individual*, std::vector<Individual, std::allocator<Individual> > >::operator--()
  1.34      7.04     0.12 253240034     0.00     0.00  __gnu_cxx::__normal_iterator<Individual*, std::vector<Individual, std::allocator<Individual> > >::base() const
  1.16      7.14     0.10 14199973     0.00     0.00  void std::__unguarded_linear_insert<__gnu_cxx::__normal_iterator<Individual*, std::vector<Individual, std::allocator<Individual> > >, __
gnu_cxx::__ops::_Val_less_iter>(__gnu_cxx::__normal_iterator<Individual*, std::vector<Individual, std::allocator<Individual> > >, __gnu_cxx::__ops::_Val_less_iter)
  1.10      7.24     0.10 25117225     0.00     0.00  void __gnu_cxx::new_allocator<Individual>::construct<Individual, Individual>(Individual*, Individual&&)
  0.99      7.33     0.09 125385363     0.00     0.00  __gnu_cxx::__normal_iterator<Individual*, std::vector<Individual, std::allocator<Individual> > >::operator++()
  0.93      7.41     0.08 46200000     0.00     0.00  Individual::Individual(Individual const&)
  0.93      7.49     0.08 32159911     0.00     0.00  mutated_genes()
  0.81      7.55     0.07 12000002     0.00     0.00  void std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> >::_M_construct<char const*>(char const*, char const*
, std::forward_iterator_tag)
  0.76      7.62     0.07 86551675     0.00     0.00  Individual* std::__addressof<Individual>(Individual&)
```

Call Graph of Serial Code

```
Call graph

granularity: each sample hit covers 4 byte(s) for 0.12% of 8.60 seconds

index % time    self  children    called     name
                                     1             main [1]
[1]     90.9    0.07    7.75       0+1       main [1]
                0.00    3.66      23/23          void std::sort<__gnu_cxx::__normal_iterator<Individual*, std::vector<Individual, std::allocator<Individual> > > >(__gnu_cxx::__normal_iterato
r<Individual*, std::vector<Individual, std::allocator<Individual> > >, __gnu_cxx::__normal_iterator<Individual*, std::vector<Individual, std::allocator<Individual> > >) [2]
                0.88    2.12 11000000/11000000     Individual::mate(Individual) [7]
                0.01    0.49 13200000/13200000     std::vector<Individual, std::allocator<Individual> >::push_back(Individual const&) [14]
                0.02    0.15 1000000/1000000     create_gnome[abi:cxx11]() [35]
                0.00    0.11 1000000/12000000     Individual::Individual(std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> >) [9]
                0.00    0.06      23/23          std::vector<Individual, std::allocator<Individual> >::~vector() [48]
                0.06    0.00 33000000/46200000     Individual::Individual(Individual const&) [44]
                0.00    0.05 1000000/1000000     std::vector<Individual, std::allocator<Individual> >::push_back(Individual&&) [52]
                0.05    0.00 45000000/205683262     Individual::~Individual() [31]
                0.05    0.00 22000000/373159911     random_num(int, int) [13]
                0.00    0.02      22/22          std::vector<Individual, std::allocator<Individual> >::operator=(std::vector<Individual, std::allocator<Individual> > const&) [70]
                0.01    0.00 11000000/11001976     std::vector<Individual, std::allocator<Individual> >::size() const [87]
                0.00    0.00       2/12000002     std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> >::basic_string<std::allocator<char> >(char const*, std::alloc
ator<char> const&) [34]
                0.00    0.00 24200069/24200069     std::vector<Individual, std::allocator<Individual> >::operator[](unsigned long) [105]
                0.00    0.00      23/23          std::vector<Individual, std::allocator<Individual> >::vector() [137]
                0.00    0.00      23/1000528     std::vector<Individual, std::allocator<Individual> >::end() [108]
                0.00    0.00      23/528         std::vector<Individual, std::allocator<Individual> >::begin() [115]
                                     1             main [1]
-----------------------------------------------
                0.00    3.66      23/23          main [1]
[2]     42.6    0.00    3.66      23         void std::sort<__gnu_cxx::__normal_iterator<Individual*, std::vector<Individual, std::allocator<Individual> > > >(__gnu_cxx::__normal_iterator<In
dividual*, std::vector<Individual, std::allocator<Individual> > >, __gnu_cxx::__normal_iterator<Individual*, std::vector<Individual, std::allocator<Individual> > >) [2]
                0.00    3.66      23/23          void std::__sort<__gnu_cxx::__normal_iterator<Individual*, std::vector<Individual, std::allocator<Individual> > >, __gnu_cxx::__ops::_Iter_le
ss_iter>(__gnu_cxx::__normal_iterator<Individual*, std::vector<Individual, std::allocator<Individual> > >, __gnu_cxx::__normal_iterator<Individual*, std::vector<Individual, std::allocator<In
dividual> > >, __gnu_cxx::__ops::_Iter_less_iter) [3]
                0.00    0.00      23/23          __gnu_cxx::__ops::__iter_less_iter() [129]
-----------------------------------------------
                0.00    3.66      23/23          void std::sort<__gnu_cxx::__normal_iterator<Individual*, std::vector<Individual, std::allocator<Individual> > > >(__gnu_cxx::__normal_iterato
r<Individual*, std::vector<Individual, std::allocator<Individual> > >, __gnu_cxx::__normal_iterator<Individual*, std::vector<Individual, std::allocator<Individual> > >) [2]
[3]     42.6    0.00    3.66      23         void std::__sort<__gnu_cxx::__normal_iterator<Individual*, std::vector<Individual, std::allocator<Individual> > >, __gnu_cxx::__ops::_Iter_less_i
ter>(__gnu_cxx::__normal_iterator<Individual*, std::vector<Individual, std::allocator<Individual> > >, __gnu_cxx::__normal_iterator<Individual*, std::vector<Individual, std::allocator<Indivi
dual> > >, __gnu_cxx::__ops::_Iter_less_iter) [3]
                0.00    3.21      23/23          void std::__introsort_loop<__gnu_cxx::__normal_iterator<Individual*, std::vector<Individual, std::allocator<Individual> > >, long, __gnu_cxx:
:__ops::_Iter_less_iter>(__gnu_cxx::__normal_iterator<Individual*, std::vector<Individual, std::allocator<Individual> > >, __gnu_cxx::__normal_iterator<Individual*, std::vector<Individual, s
td::allocator<Individual> > >, long, __gnu_cxx::__ops::_Iter_less_iter) [4]
```

Profiling Inference

From the above flat table we can clearly see functions like Individual::cal_fitness(), Individual::mate(Individual), random_num(int, int) taking a maximum amount of time like 15.35%, 10.23% and 9.48% respectively.

So, according to functional profiling if we somehow parallelise these functions we can reduce total execution time.

One more important thing we can observe is that these functions are not doing much work, the whole program is all about creating a vector of population and iterating over it again and again and finding the most suitable individual. This is very similar to array computation so we need to focus more on dividing vectors into chunks and assigning each chunk to workers/grids.

MPI Parallel Code

```
// C++ program to create target string, starting from random string using Genetic Algorithm
#include "iostream"
#include <mpi.h>
#include "vector"
#include "time.h"
#include "algorithm"
using namespace std;

bool initial_population_flag = true;
bool ten_percent_flag = false;
bool remaining_population_flag = false;
bool copy_flag = false;

#define MASTER 0
#define FROM_MASTER 1
#define FROM_WORKER 2
// Number of individuals in each generation
#define POPULATION_SIZE 171000
// #define POPULATION_SIZE 500000
int generation = 0;
// Valid Genes
const string GENES =
"`~1!2@3#4$5%6^7&8*9(0)-_+QWERTYUIOPASDFGHJKLZXCVBNMqwertyuiopasdfghjklzxcvbnm[{]}|;:'\",./?
< >";
```

```
// Target string to be generated
const string TARGET = "Random Generation..";
#define targetSize 19
// Function to generate random numbers in given range
// int random_num(int start, int end) {
//     int range = (end-start)+1;
//     int random_int = start+(rand()%range);
//     return random_int;
// }
// Create random genes for mutation
// char mutated_genes() {
//     int len = GENES.size();
//     return GENES[rand()%len];
// }
// create chromosome or string of genes
string create_gnome() {
   int len = TARGET.size();
   string gnome = "";
   for(int i=0; i<len; i++)
       gnome += GENES[rand()%len];
   return gnome;
}
// // Class representing individual in population
// class Individual {

//     public:
//         string chromosome;
//         int fitness;

//         Individual(string chromosome);
//         // Individual* mate(Individual* parent2);
//         // int cal_fitness();
// };
// Individual::Individual(string chromosome) {
//     this->chromosome = chromosome;
//     // this->fitness = cal_fitness();

   // int len = TARGET.size();
   // int offspring_fitness = 0;
   // // #pragma omp parallel for shared(TARGET, chromosome) reduction(+:fitness)
   // for(int i=0; i<len; i++) {
   //     if(chromosome[i] != TARGET[i]){
   //         offspring_fitness++;
   //     }
   // }

   // this->fitness = offspring_fitness;

// };


typedef struct Individual {
```

```
    int fitness;
    char chromosome[targetSize];

} person;




// Perform mating and produce new offspring
// Individual* Individual::mate(Individual* par2) {
//     // chromosome for offspring
//     string child_chromosome = "";
//     int len = chromosome.size();

//     for(int i = 0;i<len;i++) {
//         // random probability
//         float p = (rand()%101)/100;
//         // if prob is less than 0.45, insert gene from parent 1
//         if(p < 0.45)
//             child_chromosome += chromosome[i];
//         // if prob is between 0.45 and 0.90, insertgene from parent 2
//         else if(p < 0.90)
//             child_chromosome += par2->chromosome[i];
//         // otherwise insert random gene(mutate), for maintaining diversity
//         else
//             child_chromosome += mutated_genes();
//     }
//     // create new Individual(offspring) using generated chromosome for offspring
//     return new Individual(child_chromosome);

// };
// Calculate fitness score, it is the number of
// characters in string which differ from target string.
// int Individual::cal_fitness() {

//     int len = TARGET.size();
//     int fitness = 0;
//     // #pragma omp parallel for shared(TARGET, chromosome) reduction(+:fitness)
//     for(int i=0; i<len; i++) {
//         if(chromosome[i] != TARGET[i]){
//             fitness++;
//         }
//     }
//     return fitness;
// };
// Overloading < operator
bool compare(struct Individual ind1, struct Individual ind2) {
    return ind1.fitness < ind2.fitness;
}
// Driver code
int main(int argc, char *argv[]) {
```

```
srand(time(0));
double start, end;
int numtasks, taskid, numworkers, source, mtype, segment, aveseg, extra, offset, rc;
MPI_Status status;

MPI_Init(&argc, &argv);

MPI_Comm_rank(MPI_COMM_WORLD, &taskid);

MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
if (numtasks < 2) {
    MPI_Abort(MPI_COMM_WORLD, rc);
    exit(1);
}


char pro_name[MPI_MAX_PROCESSOR_NAME];
int name_len;
MPI_Get_processor_name(pro_name, &name_len);
numworkers = numtasks - 1;                      // numworkers are workers for process...


bool found = false;

// population size...
struct Individual population[POPULATION_SIZE];

// Otherwise generate new offsprings for new generation
struct Individual new_generation[POPULATION_SIZE];



// creating datatype for MPI to pass in send and recv..
const int nitems = 2;   // elements in struct..
MPI_Datatype myDataType;
int blocklengths[2] = {1, targetSize};
MPI_Datatype types[2] = {MPI_INT, MPI_CHAR};
MPI_Aint     offsets[2];

offsets[0] = offsetof(person, fitness);
offsets[1] = offsetof(person, chromosome);

MPI_Type_create_struct(nitems, blocklengths, offsets, types, &myDataType);
MPI_Type_commit(&myDataType);


// bool initial_population_flag = true;
// bool ten_percent_flag = false;
// bool remaining_population_flag = false;
// bool copy_flag = false;

//master task:
```

```
    if (taskid == MASTER){

        start = MPI_Wtime();

        // create initial Population.....
        aveseg = POPULATION_SIZE / numworkers;      // average segmentation for each
worker....
        extra = POPULATION_SIZE % numworkers;       // number of workers need to do extra
work..
        offset = 0;
        mtype = FROM_MASTER;


        for(int dest=1; dest<=numworkers; dest++) {

            segment = (dest <= extra) ? aveseg + 1 : aveseg;
            MPI_Send(&offset, 1, MPI_INT, dest, mtype, MPI_COMM_WORLD);
            MPI_Send(&segment, 1, MPI_INT, dest, mtype, MPI_COMM_WORLD);
            MPI_Send(&population[offset], segment, myDataType, dest, mtype, MPI_COMM_WORLD);
            offset = offset + segment;

            // MPI_Barrier(MPI_COMM_WORLD);
        }

        mtype = FROM_WORKER;
        for (int i = 1; i <= numworkers; i++) {
            source = i;
            MPI_Recv(&offset, 1, MPI_INT, source, mtype, MPI_COMM_WORLD, &status);
            MPI_Recv(&segment, 1, MPI_INT, source, mtype, MPI_COMM_WORLD, &status);
            MPI_Recv(&population[offset], segment, myDataType, source, mtype, MPI_COMM_WORLD,
&status);

            // MPI_Barrier(MPI_COMM_WORLD);

        }

        initial_population_flag = false;

        while(!found) {

            // sort the population in increasing order of fitness score
            sort(population, population+POPULATION_SIZE, compare);

            // if the individual having lowest fitness score ie. 0 then we know that we have
reached to the target and break the loop
            if(population[0].fitness <= 0){
                found = true;
                break;
            }


            ten_percent_flag = true;
```

```
            // Perform Elitism, that mean 10% of fittest population goes to the next
generation
            int s = (10*POPULATION_SIZE)/100;
            for(int i=0; i<s; i++)
                new_generation[i] = population[i];


            // From 50% of fittest population, Individuals will mate to produce offspring
            int right = (50*POPULATION_SIZE)/100;

            for(int i=s; i<POPULATION_SIZE; i++) {

                int r = rand()%(right+1);
                // Individual* parent1 = population[r];
                struct Individual parent1 = population[r];

                int offSpring_fittness = 0;


                r = rand()%(right+1);
                // Individual* parent2 = population[r];
                struct Individual parent2 = population[r];

                // chromosome for offspring
                char child_chromosome[targetSize];

                int len = TARGET.size();

                for(int k=0; k<len; k++) {
                    // random probability
                    float p = (rand()%101)/100;

                    // if prob is less than 0.45, insert gene from parent 1
                    if(p < 0.45)
                        child_chromosome[k] = parent1.chromosome[k];
                    // if prob is between 0.45 and 0.90, insertgene from parent 2
                    else if(p < 0.90)
                        child_chromosome[k] = parent2.chromosome[k];
                    // otherwise insert random gene(mutate), for maintaining diversity
                    else{
                        int range = GENES.size();
                        child_chromosome[k] = GENES[rand()%range];
                    }


                    // setting chromosome for new Individual...
                    new_generation[i].chromosome[k] = child_chromosome[k];
                    if(child_chromosome[k] != TARGET[k]){
                        offSpring_fittness++;
                    }
```

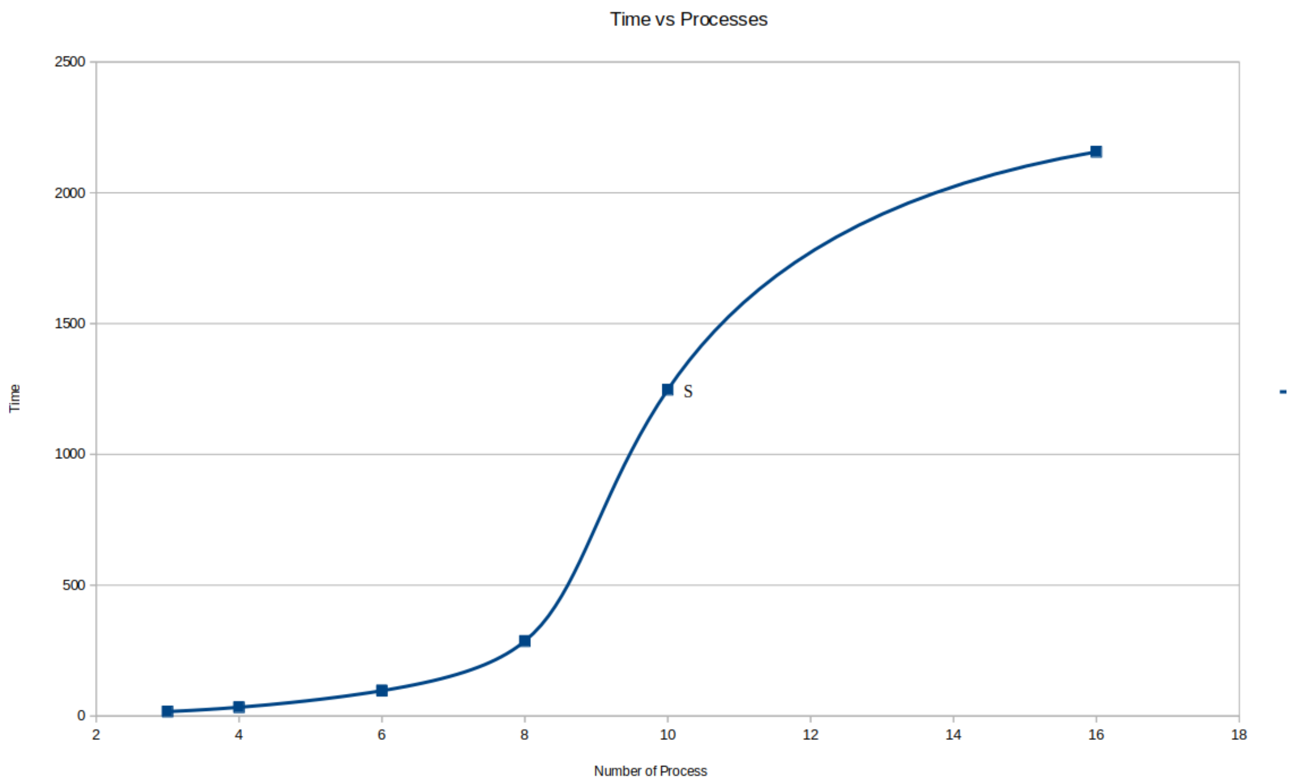
```
            }

            new_generation[i].fitness = offSpring_fittness;

        }


        // copy new_generation back to population...
        for(int i=0; i&lt;segment; i++){
            population[i] = new_generation[i];
        }

        // cout<< "Generation: " << generation << "\t";
        // cout<< "String: "<< population[0].chromosome <<"\t";
        // cout<< "Fitness: "<< population[0].fitness << "\n";
        generation++;


    }

    // cout<< "Generation: " << generation << "\t";
    // cout<< "String: "<< population[0].chromosome <<"\t";
    // cout<< "Fitness: "<< population[0].fitness << "\n\n";


    end = MPI_Wtime();
    cout << numworkers << " " << end-start << endl;

}




//Worker task:
if (taskid > MASTER) {

    // create initial population
    mtype = FROM_MASTER;
    MPI_Recv(&offset, 1, MPI_INT, MASTER, mtype, MPI_COMM_WORLD, &status);
    MPI_Recv(&segment, 1, MPI_INT, MASTER, mtype, MPI_COMM_WORLD, &status);
    MPI_Recv(&population, segment, myDataType, MASTER, mtype, MPI_COMM_WORLD, &status);

    // MPI_Barrier(MPI_COMM_WORLD);


    // work...
    for(int i=0; i&lt;segment; i++) {

        int offspring_fitness = 0;
        int len = TARGET.size();
        int geneLength = GENES.size();
```

```
            string gnome = "";
            for(int j=0; j<len; j++){
                gnome += GENES[rand()%geneLength];
                population[i].chromosome[j] = gnome[j];     // putting gnome value in
chromosome....
                if(gnome[j] != TARGET[j]){
                    offspring_fitness++;
                }
            }
            population[i].fitness = offspring_fitness;
        }

        mtype = FROM_WORKER;
        MPI_Send(&offset, 1, MPI_INT, MASTER, mtype, MPI_COMM_WORLD);
        MPI_Send(&segment, 1, MPI_INT, MASTER, mtype, MPI_COMM_WORLD);
        MPI_Send(&population, segment, myDataType, MASTER, mtype, MPI_COMM_WORLD);

        // cout << "\nInitial Population Condition complete \n";

    }

    MPI_Finalize();

    return 0;

}
```

Table

Processes	Time (seconds)
3	17.1407
4	33.5666
6	96.9177
8	286.438
10	1247.52
16	2156.65

Time vs Processes Graph



Result

Performance is not improving here because that whole code is not parallelized; only a small portion of the code is being parallelized here.

I was trying to solve this Genetic Algorithm problem same as vector addition, but the issue with MPI arries is that, my serial code is written completely on class objects and vector STL, my vector is vector of pointers, but passing of classes in MPI_send() or MPI_recv() is not posssible because this type of data type is not supported in MPI programming for that i had to change the hole code into structure based and simple array based approach, but to pass struct we need to define data type in MPI.
This issue could be solved using defining your own data type in MPI.
I have tested each block separately and all of them work perfectly but for some reason it's going into a blocking state when I run them together, since the code consists of the loop which calls the workers again and again it's quite tough to figure out the blocking stage.

I have made several changes in data types and segmentations for each worker and their sending and receiving parameters. After a lot of iterations I have made parts of the code parallelized.

[P.S.]: These values are computed on a single process not on a cluster. That's why one expected reasoning could be for such exponential time growth.

Inference

From the above Processes vs Time graph, we can clearly see that the performance is decreasing with the increasing number of processors.
The only possible reason we can think of that is previously when we were doing in OpenMP computations were happening in single computer there was no network dependency only communication overhead was the issue, but in this case, we are doing distributed computing which means computation has to depend on network overhead which is very slow as compared to previous case that’s why program getting too slow with increase number of processors(computers).
In this case time is increasing exponentially because we are doing this in a single processor with different hosts.

Cuda Parallelization

Serial C++ Code

```
%%cu

// C++ program to create target string, starting from random string using Genetic Algorithm
#include "iostream"
#include "vector"
#include "time.h"
#include "algorithm"
using namespace std;
// Number of individuals in each generation
#define POPULATION_SIZE 1000
// #define POPULATION_SIZE 500000
int generation = 0;
// Valid Genes
const string GENES =
"`~1!2@3#4$5%6^7&8*9(0)-_+QWERTYUIOPASDFGHJKLZXCVBNMqwertyuiopasdfghjklzxcvbnm[{]}|;:'\",./?
< >";

// Target string to be generated
const string TARGET = "Random Generation...";

// Create random genes for mutation
char mutated_genes() {
   int len = GENES.size();
   return GENES[rand()%len];
}
// create chromosome or string of genes
string create_gnome() {
   int len = TARGET.size();
   string gnome = "";
   for(int i=0; i<len; i++)
       gnome += mutated_genes();
   return gnome;
}
// Class representing individual in population
class Individual {

   public:
       string chromosome;
       int fitness;

       Individual(string chromosome);
};
Individual::Individual(string chromosome) {
   this->chromosome = chromosome;
```

```
    int len = TARGET.size();
    int offspring_fitness = 0;
    // #pragma omp parallel for shared(TARGET, chromosome) reduction(+:fitness)
    for(int i=0; i<len; i++) {
        if(chromosome[i] != TARGET[i]){
            offspring_fitness++;
        }
    }
    this->fitness = offspring_fitness;
};
// Overloading < operator
bool compare(Individual* ind1, Individual* ind2) {
    return ind1->fitness < ind2->fitness;
}
// Driver code
int main() {

    double start_time, end_time;
    int threads;
    srand(time(0));
    vector<Individual*> population(POPULATION_SIZE);
    bool found = false;
    // create initial population
    for(int i=0; i<POPULATION_SIZE; i++) {
        string gnome = create_gnome();
        population[i] = new Individual(gnome);
    }

    while(!found) {

        // sort the population in increasing order of fitness score
        sort(population.begin(), population.end(), compare);

        // if the individual having lowest fitness score ie. 0 then we know that we have
reached to the target and break the loop
        if(population[0]->fitness <= 0){
            found = true;
            break;
        }

        // Otherwise generate new offsprings for new generation
        vector<Individual*> new_generation(POPULATION_SIZE);

        // Perform Elitism, that mean 10% of fittest population goes to the next generation
        int s = (10*POPULATION_SIZE)/100;

        for(int i=0; i<s; i++){
            new_generation[i] = population[i];
        }

        // From 50% of fittest population, Individuals will mate to produce offspring
        int right = (50*POPULATION_SIZE)/100;
```

```
        for(int i=s; i<POPULATION_SIZE; i++) {

            // int r = random_num(0, right);
            int r = rand()%(right+1);
            Individual* parent1 = population[r];

            // r = random_num(0, right);
            r = rand()%(right+1);
            Individual* parent2 = population[r];

            // chromosome for offspring
            string child_chromosome = "";
            string chromosome = parent1->chromosome;
            int len = chromosome.size();

            for(int i = 0;i<len;i++) {
                // random probability
                float p = (rand()%101)/100;

                // if prob is less than 0.45, insert gene from parent 1
                if(p < 0.45)
                    child_chromosome += chromosome[i];
                // if prob is between 0.45 and 0.90, insertgene from parent 2
                else if(p < 0.90)
                    child_chromosome += parent2->chromosome[i];
                // otherwise insert random gene(mutate), for maintaining diversity
                else
                    child_chromosome += mutated_genes();
            }

            // create new Individual(offspring) using generated chromosome for offspring
            Individual* offspring = new Individual(child_chromosome);

            new_generation[i] = offspring;
        }

        population = new_generation;
        cout<< "Generation: " << generation << "\t";
        cout<< "String: "<< population[0]->chromosome <<"\t";
        cout<< "Fitness: "<< population[0]->fitness << "\n";

        generation++;

    }

    cout<< "Generation: " << generation << "\t";
    cout<< "String: "<< population[0]->chromosome <<"\t";
    cout<< "Fitness: "<< population[0]->fitness << "\n";

  return 0;
}
```

Flat Profile of Serial Code

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 15.35      1.32     1.32 12000000     0.00     0.00  Individual::cal_fitness()
 10.23      2.20     0.88 11000000     0.00     0.00  Individual::mate(Individual)
  9.48      3.02     0.81 373159911     0.00     0.00  random_num(int, int)
  5.70      3.50     0.49                             _init
  5.29      3.96     0.46 725254058     0.00     0.00  __gnu_cxx::__normal_iterator<Individual*, std::vector<Individual, std::allocator<Individual> > >::operator*() const
  4.13      4.32     0.35 246659903     0.00     0.00  operator<(Individual const&, Individual const&)
  4.13      4.67     0.35 108166060     0.00     0.00  std::enable_if<std::__and_<std::__not_<std::__is_tuple_like<Individual> >, std::is_move_constructible<Individual>, std::is_move_assigna
ble<Individual> >::value, void>::type std::swap<Individual>(Individual&, Individual&)
  3.78      5.00     0.33 232457779     0.00     0.00  bool __gnu_cxx::__ops::_Iter_less_iter::operator()<__gnu_cxx::__normal_iterator<Individual*, std::vector<Individual, std::allocator<Ind
ividual> > >, __gnu_cxx::__normal_iterator<Individual*, std::vector<Individual, std::allocator<Individual> > > >(__gnu_cxx::__normal_iterator<Individual*, std::vector<Individual, std::alloca
tor<Individual> > >, __gnu_cxx::__normal_iterator<Individual*, std::vector<Individual, std::allocator<Individual> > >) const
  3.49      5.29     0.30  1284267     0.00     0.00  __gnu_cxx::__normal_iterator<Individual*, std::vector<Individual, std::allocator<Individual> > > std::__unguarded_partition<__gnu_cxx::_
_normal_iterator<Individual*, std::vector<Individual, std::allocator<Individual> > >, __gnu_cxx::__ops::_Iter_less_iter>(__gnu_cxx::__normal_iterator<Individual*, std::vector<Individual, std
::allocator<Individual> > >, __gnu_cxx::__normal_iterator<Individual*, std::vector<Individual, std::allocator<Individual> > >, __gnu_cxx::__normal_iterator<Individual*, std::vector<Individua
l, std::allocator<Individual> > >, __gnu_cxx::__ops::_Iter_less_iter)
  3.37      5.58     0.29 230534256     0.00     0.00  Individual::operator=(Individual&&)
  2.85      5.83     0.24 147483262     0.00     0.00  Individual::Individual(Individual&&)
  2.79      6.07     0.24 378017518     0.00     0.00  std::remove_reference<Individual&>::type&& std::move<Individual&>(Individual&)
  2.56      6.29     0.22 205683262     0.00     0.00  Individual::~Individual()
  2.21      6.48     0.19                             void std::__partial_sort<__gnu_cxx::__normal_iterator<Individual*, std::vector<Individual, std::allocator<Individual> > >, __gnu_cxx::__
ops::_Iter_less_iter>(__gnu_cxx::__normal_iterator<Individual*, std::vector<Individual, std::allocator<Individual> > >, __gnu_cxx::__normal_iterator<Individual*, std::vector<Individual, std:
:allocator<Individual> > >, __gnu_cxx::__normal_iterator<Individual*, std::vector<Individual, std::allocator<Individual> > >, __gnu_cxx::__ops::_Iter_less_iter)
  1.98      6.65     0.17 108166060     0.00     0.00  void std::iter_swap<__gnu_cxx::__normal_iterator<Individual*, std::vector<Individual, std::allocator<Individual> > >, __gnu_cxx::__norm
al_iterator<Individual*, std::vector<Individual, std::allocator<Individual> > > >(__gnu_cxx::__normal_iterator<Individual*, std::vector<Individual, std::allocator<Individual> > >, __gnu_cxx:
:__normal_iterator<Individual*, std::vector<Individual, std::allocator<Individual> > >)
  1.74      6.80     0.15 108166060     0.00     0.00  bool __gnu_cxx::operator< <Individual*, std::vector<Individual, std::allocator<Individual> > >(__gnu_cxx::__normal_iterator<Individual*
, std::vector<Individual, std::allocator<Individual> > > const&, __gnu_cxx::__normal_iterator<Individual*, std::vector<Individual, std::allocator<Individual> > > const&)
  1.51      6.93     0.13 130737308     0.00     0.00  __gnu_cxx::__normal_iterator<Individual*, std::vector<Individual, std::allocator<Individual> > >::operator--()
  1.34      7.04     0.12 253240034     0.00     0.00  __gnu_cxx::__normal_iterator<Individual*, std::vector<Individual, std::allocator<Individual> > >::base() const
  1.16      7.14     0.10 14199973     0.00     0.00  void std::__unguarded_linear_insert<__gnu_cxx::__normal_iterator<Individual*, std::vector<Individual, std::allocator<Individual> > >, __
gnu_cxx::__ops::_Val_less_iter>(__gnu_cxx::__normal_iterator<Individual*, std::vector<Individual, std::allocator<Individual> > >, __gnu_cxx::__ops::_Val_less_iter)
  1.10      7.24     0.10 25117225     0.00     0.00  void __gnu_cxx::new_allocator<Individual>::construct<Individual, Individual>(Individual*, Individual&&)
  0.99      7.33     0.09 125385363     0.00     0.00  __gnu_cxx::__normal_iterator<Individual*, std::vector<Individual, std::allocator<Individual> > >::operator++()
  0.93      7.41     0.08 46200000     0.00     0.00  Individual::Individual(Individual const&)
  0.93      7.49     0.08 32159911     0.00     0.00  mutated_genes()
  0.81      7.55     0.07 12000002     0.00     0.00  void std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> >::_M_construct<char const*>(char const*, char const*
, std::forward_iterator_tag)
  0.76      7.62     0.07 86551675     0.00     0.00  Individual* std::__addressof<Individual>(Individual&)
```

Call Graph of Serial Code

```
Call graph

granularity: each sample hit covers 4 byte(s) for 0.12% of 8.60 seconds

index % time    self  children    called     name
                                     1             main [1]
[1]     90.9    0.07    7.75     0+1         main [1]
                0.00    3.66      23/23          void std::sort<__gnu_cxx::__normal_iterator<Individual*, std::vector<Individual, std::allocator<Individual> > > >(__gnu_cxx::__normal_iterato
r<Individual*, std::vector<Individual, std::allocator<Individual> > >, __gnu_cxx::__normal_iterator<Individual*, std::vector<Individual, std::allocator<Individual> > >) [2]
                0.88    2.12 11000000/11000000     Individual::mate(Individual) [7]
                0.01    0.49 13200000/13200000     std::vector<Individual, std::allocator<Individual> >::push_back(Individual const&) [14]
                0.02    0.15 1000000/1000000     create_gnome[abi:cxx11]() [35]
                0.00    0.11 1000000/12000000     Individual::Individual(std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> >) [9]
                0.00    0.06      23/23          std::vector<Individual, std::allocator<Individual> >::~vector() [48]
                0.06    0.00 33000000/46200000     Individual::Individual(Individual const&) [44]
                0.00    0.05 1000000/1000000     std::vector<Individual, std::allocator<Individual> >::push_back(Individual&&) [52]
                0.05    0.00 45000000/205683262      Individual::~Individual() [31]
                0.05    0.00 22000000/373159911      random_num(int, int) [13]
                0.00    0.02      22/22          std::vector<Individual, std::allocator<Individual> >::operator=(std::vector<Individual, std::allocator<Individual> > const&) [70]
                0.01    0.00 11000000/11001976     std::vector<Individual, std::allocator<Individual> >::size() const [87]
                0.00    0.00       2/12000002     std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> >::basic_string<std::allocator<char> >(char const*, std::alloc
ator<char> const&) [34]
                0.00    0.00 24200069/24200069     std::vector<Individual, std::allocator<Individual> >::operator[](unsigned long) [105]
                0.00    0.00      23/23          std::vector<Individual, std::allocator<Individual> >::vector() [137]
                0.00    0.00      23/1000528     std::vector<Individual, std::allocator<Individual> >::end() [108]
                0.00    0.00      23/528         std::vector<Individual, std::allocator<Individual> >::begin() [115]
                                     1             main [1]
-----------------------------------------------
                0.00    3.66      23/23          main [1]
[2]     42.6    0.00    3.66      23         void std::sort<__gnu_cxx::__normal_iterator<Individual*, std::vector<Individual, std::allocator<Individual> > > >(__gnu_cxx::__normal_iterator<In
dividual*, std::vector<Individual, std::allocator<Individual> > >, __gnu_cxx::__normal_iterator<Individual*, std::vector<Individual, std::allocator<Individual> > >) [2]
                0.00    3.66      23/23          void std::__sort<__gnu_cxx::__normal_iterator<Individual*, std::vector<Individual, std::allocator<Individual> > >, __gnu_cxx::__ops::_Iter_le
ss_iter>(__gnu_cxx::__normal_iterator<Individual*, std::vector<Individual, std::allocator<Individual> > >, __gnu_cxx::__normal_iterator<Individual*, std::vector<Individual, std::allocator<In
dividual> > >, __gnu_cxx::__ops::_Iter_less_iter) [3]
                0.00    0.00      23/23          __gnu_cxx::__ops::__iter_less_iter() [129]
-----------------------------------------------
                0.00    3.66      23/23          void std::sort<__gnu_cxx::__normal_iterator<Individual*, std::vector<Individual, std::allocator<Individual> > > >(__gnu_cxx::__normal_iterato
r<Individual*, std::vector<Individual, std::allocator<Individual> > >, __gnu_cxx::__normal_iterator<Individual*, std::vector<Individual, std::allocator<Individual> > >) [2]
[3]     42.6    0.00    3.66      23         void std::__sort<__gnu_cxx::__normal_iterator<Individual*, std::vector<Individual, std::allocator<Individual> > >, __gnu_cxx::__ops::_Iter_less_i
ter>(__gnu_cxx::__normal_iterator<Individual*, std::vector<Individual, std::allocator<Individual> > >, __gnu_cxx::__normal_iterator<Individual*, std::vector<Individual, std::allocator<Indivi
dual> > >, __gnu_cxx::__ops::_Iter_less_iter) [3]
                0.00    3.21      23/23          void std::__introsort_loop<__gnu_cxx::__normal_iterator<Individual*, std::vector<Individual, std::allocator<Individual> > >, long, __gnu_cxx:
:__ops::_Iter_less_iter>(__gnu_cxx::__normal_iterator<Individual*, std::vector<Individual, std::allocator<Individual> > >, __gnu_cxx::__normal_iterator<Individual*, std::vector<Individual, s
td::allocator<Individual> > >, long, __gnu_cxx::__ops::_Iter_less_iter) [4]
```

Profiling Inference

From the above flat table we can clearly see functions like Individual::cal_fitness(), Individual::mate(Individual), random_num(int, int) taking a maximum amount of time like 15.35%, 10.23% and 9.48% respectively.
So, according to functional profiling if we somehow parallelise these functions we can reduce total execution time.
One more important thing we can observe is that these functions are not doing much work, the whole program is all about creating a vector of population and iterating over it again and again and finding the most suitable individual. This is very similar to array computation so we need to focus more on dividing vectors into chunks and assigning each chunk to workers/grids.

Parallel Cuda Code

```
%%cu

// C++ program to create target string, starting from random string using
Genetic Algorithm
#include "bits/stdc++.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <cuda.h>
#include <curand.h>
#include <curand_kernel.h>
using namespace std;
// Number of individuals in each generation
#define POPULATION_SIZE 1
// #define POPULATION_SIZE 500000
int generation = 0;
// Valid Genes
#define GENES
"`~1!2@3#4$5%6^7&8*9(0)-_+QWERTYUIOPASDFGHJKLZXCVBNMqwertyuiopasdfghjklzxcv
bnm[{]}|;:'\",./?< >"
#define genesSize 93
```

```
// Target string to be generated
#define TARGET "Random Generation..."
#define targetSize 20


// // Create random genes for mutation
// char mutated_genes() {
//     int len = GENES.size();
//     return GENES[rand()%len];
// }
// // create chromosome or string of genes
// string create_gnome() {
//     int len = TARGET.size();
//     string gnome = "";
//     for(int i=0; i<len; i++)
//         gnome += mutated_genes();
//     return gnome;
// }
// Class representing individual in population
class Individual {

   public:
       int fitness;
       char chromosome[targetSize];

       __device__ __host__ Individual(char *chromosome);
};
Individual::Individual(char *chromosome) {
   // this->chromosome = chromosome;
   for (int i=0; chromosome[i] != '\0'; i++) {
       this->chromosome[i] = chromosome[i];
   }

   // int len = TARGET.size();
   int offspring_fitness = 0;
   for(int i=0; i<targetSize; i++) {
       if(chromosome[i] != TARGET[i]){
           offspring_fitness++;
       }
   }
   this->fitness = offspring_fitness;
};
// Overloading < operator
```

```
bool compare(Individual* ind1, Individual* ind2) {
   return ind1->fitness < ind2->fitness;
}

__device__ float generate(curandState* globalState, int ind) {
   //int ind = threadIdx.x;
   curandState localState = globalState[ind];
   float RANDOM = curand_uniform( &localState );
   globalState[ind] = localState;
   return RANDOM;
}

__global__ void setup_kernel ( curandState * state, unsigned long seed ) {
   int id = threadIdx.x;
   curand_init ( seed, id, 0, &state[id] );
}


// curandState* globalState for generating random...
__global__ void population_kernel(Individual **population, curandState
*globalState) {

   // printf("Hello..");

   int index = threadIdx.x + blockIdx.x * 5;       // M is number of
blocks..

   int number;
   int id = threadIdx.x + blockIdx.x * blockDim.x;

   // create initial population

   // for(int i=0; i<POPULATION_SIZE; i++) {
   //     char gnome[targetSize];
   //     for(int j=0; j<targetSize; j++){
   //         number = generate(globalState, id)*genesSize;
   //         gnome[j] = GENES[number];
   //     }
    //     population[i] = new Individual(gnome);
   //     // for(int j=0; j<targetSize; j++){
   //     //     printf("%c", population[i]->chromosome[j]);
   //     // }
   //     // printf(" --- Fitness %d\n", population[i]->fitness);
```

```
    // }
    // // printf("\nComplete., %d", population[0]->fitness);


    if(index < POPULATION_SIZE){

        char gnome[targetSize];
        for(int j=0; j<targetSize; j++){
            number = generate(globalState, id)*genesSize;
            gnome[j] = GENES[number];
        }
         population[index] = new Individual(gnome);
        for(int j=0; j<targetSize; j++){
            printf("%c", population[index]->chromosome[j]);
        }
        printf(" --- Fitness %d\n", population[index]->fitness);

    }

    printf("Completed...\n");



}



// Driver code
int main() {

    // cout << "\nfine";

    int size = POPULATION_SIZE * sizeof(Individual);
    // Individual **population;           // host copies        ////
    Individual* population[size];

    Individual **d_population;          // host copies

    // allocate space for host...
    // population = (Individual **)malloc(size);                /////

    // allocate space for device copies
    cudaMalloc((void **)&d_population, size);
```

```
curandState* devStates;
cudaMalloc (&devStates, POPULATION_SIZE * sizeof(curandState));
srand(time(0));
/** ADD THESE TWO LINES **/
int seed = rand();
setup_kernel<<<2, 5>>>(devStates,seed);
/** END ADDITION **/

// Launch kernel on GPU with N blocks
population_kernel<<<1,1>>>(d_population, devStates);
cudaDeviceSynchronize();

// copy result back to host..
cudaMemcpy(population, d_population, 2&size, cudaMemcpyDeviceToHost);

cout << "\nfine..\n";
cout << "fine2";
// cout << "Value " << population[0]->fitness;

// for(int index=0; index<POPULATION_SIZE; index++){

//     cout << "Inside..\n";
//     for(int j=0; j<targetSize; j++){
//         cout << population[index]->chromosome[j];
//     }
//     cout << " --- Fitness " << population[index]->fitness << endl;

// }




// int threads;
// bool found = false;
// // create initial population
// for(int i=0; i<POPULATION_SIZE; i++) {
//     // string gnome = create_gnome();

//     int len = targetSize;
//     string gnome = "";
//     for(int i=0; i<len; i++)
//         gnome += GENES[rand()%genesSize];
```

```
   //     // population[i] = new Individual(gnome);
   // }


   // while(!found) {

   //     // sort the population in increasing order of fitness score
   //     sort(population.begin(), population.end(), compare);

   //     // if the individual having lowest fitness score ie. 0 then we
know that we have reached to the target and break the loop
   //     if(population[0]->fitness <= 0){
   //         found = true;
   //         break;
   //     }

   //     // Otherwise generate new offsprings for new generation
   //     vector<Individual*> new_generation(POPULATION_SIZE);

   //     // Perform Elitism, that mean 10% of fittest population goes to
the next generation
   //     int s = (10*POPULATION_SIZE)/100;

   //     for(int i=0; i<s; i++){
   //         new_generation[i] = population[i];
   //     }

   //     // From 50% of fittest population, Individuals will mate to
produce offspring
   //     int right = (50*POPULATION_SIZE)/100;

   //     for(int i=s; i<POPULATION_SIZE; i++) {

   //         // int r = random_num(0, right);
   //         int r = rand()%(right+1);
   //         Individual* parent1 = population[r];

   //         // r = random_num(0, right);
   //         r = rand()%(right+1);
   //         Individual* parent2 = population[r];
```

```
   //          // chromosome for offspring
   //          string child_chromosome = "";
   //          string chromosome = parent1->chromosome;
   //          int len = chromosome.size();

   //          for(int i = 0;i<len;i++) {
   //              // random probability
   //              float p = (rand()%101)/100;

   //              // if prob is less than 0.45, insert gene from parent 1
   //              if(p < 0.45)
   //                  child_chromosome += chromosome[i];
   //              // if prob is between 0.45 and 0.90, insertgene from
parent 2
   //              else if(p < 0.90)
   //                  child_chromosome += parent2->chromosome[i];
   //              // otherwise insert random gene(mutate), for maintaining
diversity
   //              else
   //                  child_chromosome += mutated_genes();
   //          }

   //          // create new Individual(offspring) using generated
chromosome for offspring
   //          Individual* offspring = new Individual(child_chromosome);

   //          new_generation[i] = offspring;
   //      }

   //      population = new_generation;
   //      cout<< "Generation: " << generation << "\t";
   //      cout<< "String: "<< population[0]->chromosome <<"\t";
   //      cout<< "Fitness: "<< population[0]->fitness << "\n";

   //      generation++;
   // }

   // cout<< "Generation: " << generation << "\t";
   // cout<< "String: "<< population[0]->chromosome <<"\t";
   // cout<< "Fitness: "<< population[0]->fitness << "\n";
  return 0;
}
```

Results

Result is not returning back to the host from the device.
Current kernel code is working fine and giving its desired correct output, but for some reason the result is not being returned to the host.

Code is working with Class Objects and c++ vectors or strings but the cuda kernel doesn't support classes and vectors and strings in the kernel for that I have to change the whole code according to kernel oriented C.
As stated in profiling, code is very similar to vector computation for random number generation is very important but unfortunately cuda kernel doesn't support that also, i have to manually write different kernels to create a random number based on the current threads and blocks number to generate a random integer.

I am creating host and device copies of the initial population array which is a double pointer i.e an array containing pointers. It is correctly being passed in the population kernel and doing its computation on array but for some reason it is not copying its new updated array back to its host copy. I have tried many different solutions found in internet sources like changing the data type and pointer type but nothing is working.
I tried to change data type from double pointer to array pointer. In the case of array pointer i am getting expected output but for my problem statement i need to work with the class/struct but cuda kernel doesn't support that. I have to use classes and to use a class double pointer is required.

Inference

Problem statement is heavily dependent of array computations, if we are able to divide the each array(population generation) in equal segments and pass it to GPU for computation then we can decrease our time approximately N fold, because each individual is independent to each other and there is no dependency among them so there should not be any stall or dependency to each grid in GPU.