Project Report

Parallel Genetic Algorithm for Finding Roots of Complex Polynomial Equations using MPI Master Slave Method

Submitted by
Arushi Jain (160001008)
Bitan Paul (160001015)
(Group 10)

Computer Science and Engineering
3rd year

Under the Guidance of
Dr. Surya Prakash

Department of Computer Science and Engineering
Indian Institute of Technology Indore
Autumn 2018

Problem Statement

We wish to find the roots of a complex polynomial equation iteratively by taking an initial guess and computing step by step approximations which finally converge to a single solution over time.

We solve this problem by creating a parallel genetic algorithm using MPI Master Slave Method.

Genetic Algorithms are an effective method to simulate the process of natural selection. It can find satisfied solutions for some optimization problems which are difficult to solve with traditional methods such as exhaustive search and analytic method. When Genetic Algorithms are adopted on a parallel or distributed system, it can speed up the pace of solving the problems and get better solutions.

Genetic Algorithms - An Introduction

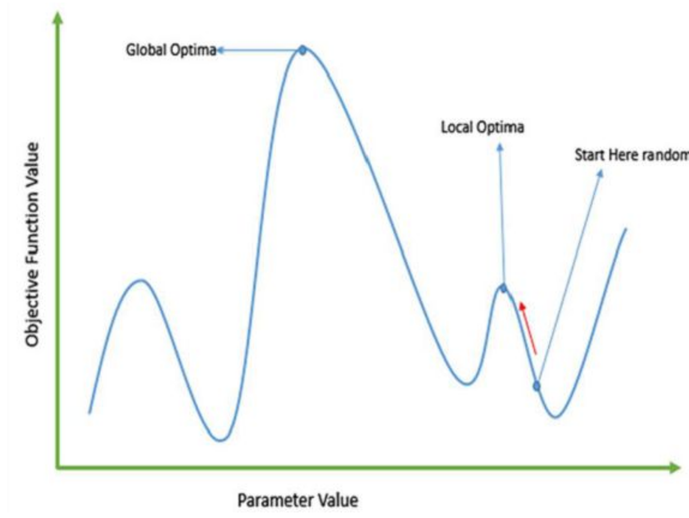
A **genetic algorithm** is a search heuristic that is inspired by Charles Darwin’s theory of natural evolution. This algorithm reflects the process of natural selection where the fittest individuals are selected for reproduction in order to produce offspring of the next generation.

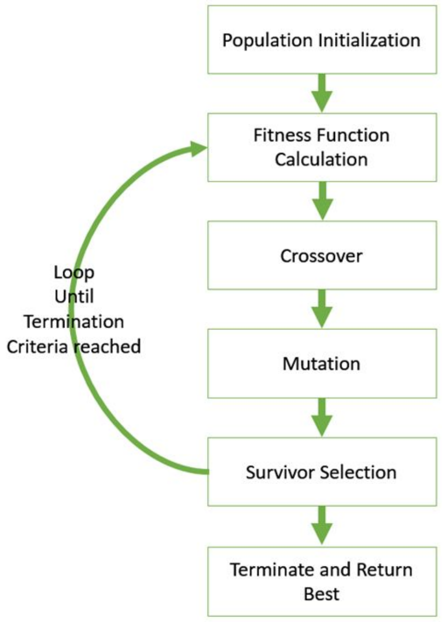
The process of natural selection starts with the selection of fittest individuals from a population. They produce offspring which inherit the characteristics of the parents and will be added to the next generation. If parents have better fitness, their offspring will be better than parents and have a better chance of surviving. This process keeps on iterating and at the end, a generation with the fittest individuals will be found.

It has the following phases:

1. Initial Population: The process begins with a set of individuals which is called a **Population**. Each individual is a solution to the problem you want to solve.
2. Fitness Function**:** The **fitness function** determines how fit an individual is (the ability of an individual to compete with other individuals). It gives a **fitness score** to each individual. The probability that an individual will be selected for reproduction is based on its fitness score.
3. Selection**:** The idea of the **selection** phase is to select the fittest individuals and let them pass their genes to the next generation. Two pairs of individuals (**parents**) are selected based on their fitness scores. Individuals with high fitness usually have more chance to be selected for reproduction.
4. Crossover: **Crossover** is the most significant phase in a genetic algorithm. For each pair of parents to be mated, a **crossover point** is chosen at random from within the genes. **Offspring** are created by exchanging the genes of parents among themselves until the crossover point is reached.
5. Mutation**:** In certain new offspring formed, some of their genes can be subjected to a **mutation** with a low random probability. This implies that some of the bits in the bit string can be flipped.
6. Termination**:** The algorithm terminates if the population has converged (does not produce offspring which are significantly different from the previous generation). Then it is said that the genetic algorithm has provided a set of solutions to our problem.

The population has a fixed size. As new generations are formed, individuals with least fitness die, providing space for new offspring. The sequence of phases is repeated to produce individuals in each new generation which are better than the previous generation.




Population Initialization
Fitness Function Calculation
Crossover
Mutation
Survivor Selection
Terminate and Return Best
Loop Until Termination Criteria reached

MPI Master Slave

MPI (Message Passing Interface) is a specification for the developers and users of message passing libraries. By itself, it is NOT a library - but rather the specification of what such a library should be. MPI primarily addresses the message-passing parallel programming model: data is moved from the address space of one process to that of another process through cooperative operations on each process.

Message passing is well-suited to handling computations where a task is divided up into subtasks, with most of the processes used to compute the subtasks and a few processes (often just one process) managing the tasks. The manager is called the "master" and the others the "workers" or the "slaves".

MPI accomplishes algorithm parallelization which aims towards speeding up a problem by running subtasks simultaneously as mentioned above. The main goals of the parallelization are to achieve more speed-up. In order to achieve this, we need to take care of three separate problems.

1. Distributing the work among the processes (running on different processors) more evenly.
   If the distribution is uneven, then this will cause some processes to run a shorter time while the whole running time will be dominated by the slower processes.
2. Locality of the data
   For shared memory systems, the possibility of reading any data at any time hides away the real problem of how far the data is. Failing to notice this problem may cause a shared memory algorithm run slower by magnitudes.

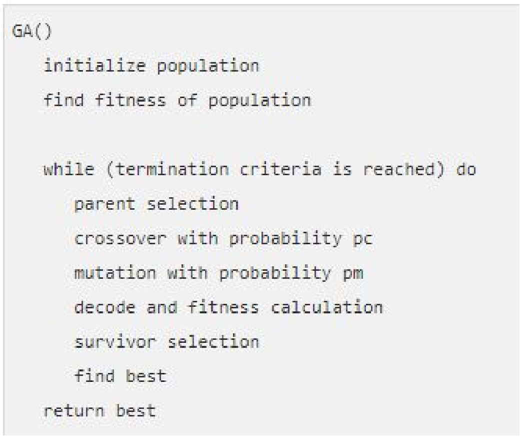
3. Overhead of the algorithm
   This one consists of that part which must be done in a serial way and the cost of the parallelization itself.

Our parallel algorithm uses MPI in the context of data parallelism. Data parallelism is when a problem is divided by data, where each subtask performs the same operations on their data and the master process generally, the process the subtasks originate from takes care of the data distribution in the subtasks and bring the data together after the termination of the sub-processes.

We define master and worker processes separately with both of them having well-defined roles in the algorithm. All the worker processes run the same code but each worker process gets populated with their own set of data. They run the same tasks on different data synchronously. The master process is the one responsible for the synchronicity of the worker processes.

Approach

1. We take the polynomial (f), population size (P), accepted error (ae), mutation rate (mr), mutation radius (ms), starting radius (sr) and the number of threads as input.
2. Using starting radius, we use Real Uniform Distribution to initialize population P with real and imaginary parts within the starting radius, thus causing the search to be unbiased.
3. Individual members of the population are evaluated to find the objective function value and mapped into a fitness function. **The closer the value of the function to zero, the fitter the individual.** Thus fitness function is inversely proportional to fitness.
4. We then sort the population in increasing order of fitness with the first individual being the fittest. We check whether the fittest solution is within the accepted error range. If yes, we terminate, else we loop until we reach a solution.
5. Reproduction is done by selecting two parents based on some appropriate method (Roulette Wheel selection or Elitism) which crossover using either a symmetric trisection method or uniform probability distribution.
6. Finally, mutation can occur in each offspring based on some probability mr up to a certain radius ms. This is how each generation will be better than the previous in proximity to the actual roots of the equation.



```
GA()
   initialize population
   find fitness of population

   while (termination criteria is reached) do
      parent selection
      crossover with probability pc
      mutation with probability pm
      decode and fitness calculation
      survivor selection
      find best
   return best
```

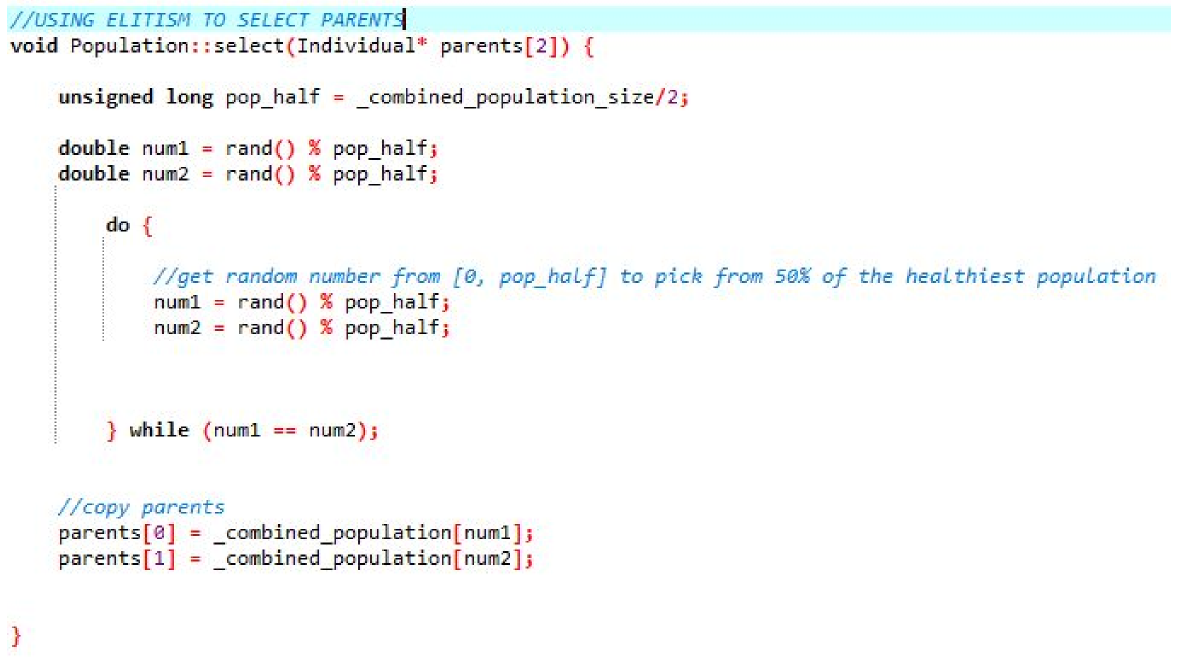
Finally, we will obtain the fittest individual that is a root of the polynomial.

Note: Multiple roots may exist for a complex polynomial equation. Our algorithm finds a single root that eventually converges to a single value based on how the solutions evolve. In earlier generations, we might see individuals closer to other roots of the equation as well. However, with each passing generation, the population must finally converge to a single solution before termination.

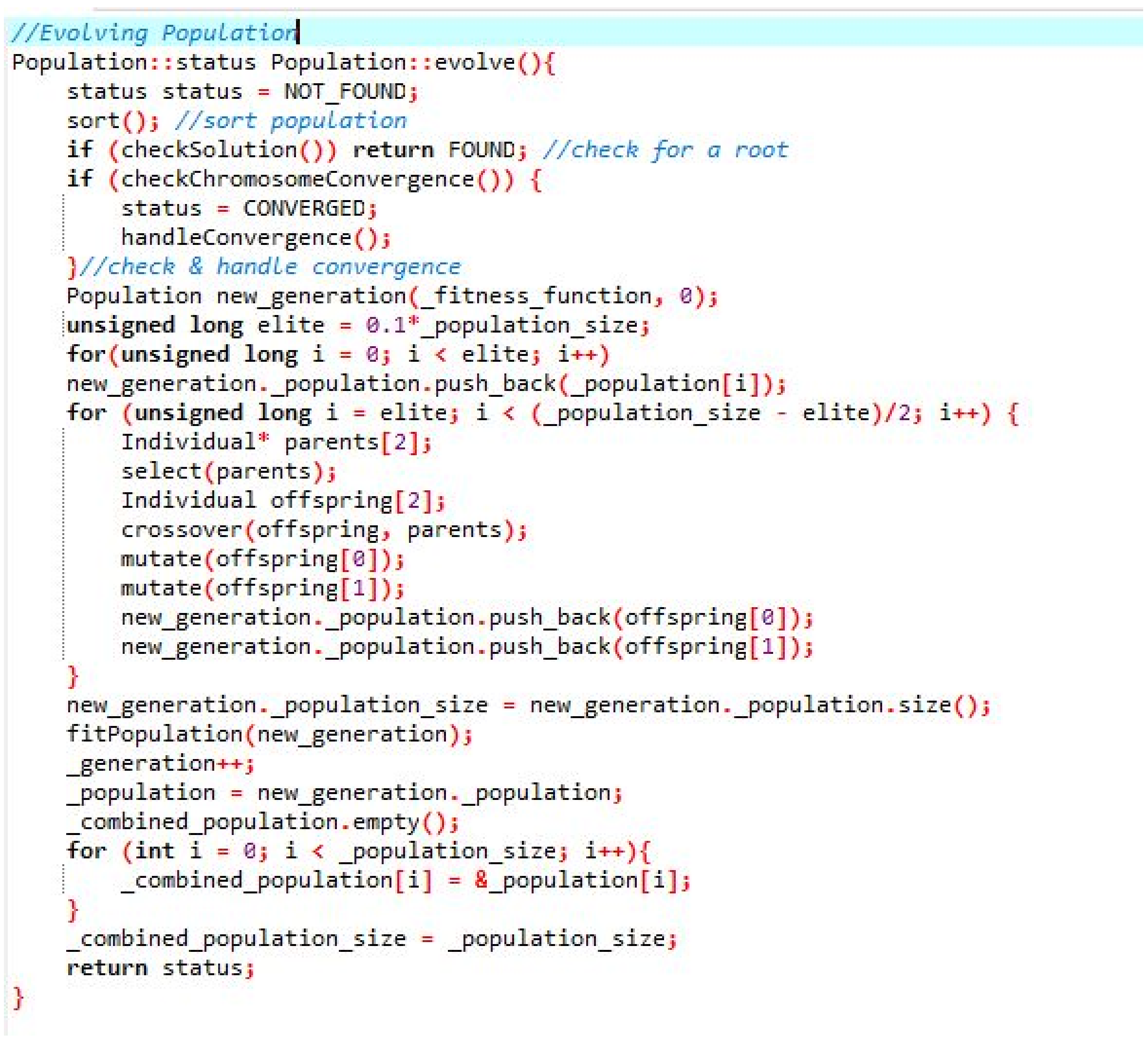
Variations in Genetic Algorithm Evolution

A. **Parent Selection**

1. **Elitism**
   In this technique, we are directly retaining the top 10% of the fittest population in the next generation without any mutations. The parents for the remaining 90% of the generation are chosen randomly using from the 50% of the healthiest population (while making sure that a parent doesn't mate with itself.)

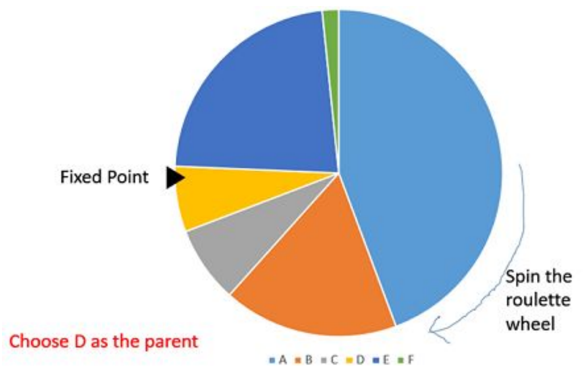


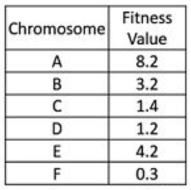
```
//USING ELITISM TO SELECT PARENTS
void Population::select(Individual* parents[2]) {

    unsigned long pop_half = _combined_population_size/2;

    double num1 = rand() % pop_half;
    double num2 = rand() % pop_half;

        do {

            //get random number from [0, pop_half] to pick from 50% of the healthiest population
            num1 = rand() % pop_half;
            num2 = rand() % pop_half;



        } while (num1 == num2);

    //copy parents
    parents[0] = _combined_population[num1];
    parents[1] = _combined_population[num2];


}
```



```
//Evolving Population
Population::status Population::evolve(){
    status status = NOT_FOUND;
    sort(); //sort population
    if (checkSolution()) return FOUND; //check for a root
    if (checkChromosomeConvergence()) {
        status = CONVERGED;
        handleConvergence();
    }//check & handle convergence
    Population new_generation(_fitness_function, 0);
    unsigned long elite = 0.1*_population_size;
    for(unsigned long i = 0; i < elite; i++)
    new_generation._population.push_back(_population[i]);
    for (unsigned long i = elite; i < (_population_size - elite)/2; i++) {
        Individual* parents[2];
        select(parents);
        Individual offspring[2];
        crossover(offspring, parents);
        mutate(offspring[0]);
        mutate(offspring[1]);
        new_generation._population.push_back(offspring[0]);
        new_generation._population.push_back(offspring[1]);
    }
    new_generation._population_size = new_generation._population.size();
    fitPopulation(new_generation);
    _generation++;
    _population = new_generation._population;
    _combined_population.empty();
    for (int i = 0; i < _population_size; i++){
        _combined_population[i] = &_population[i];
    }
    _combined_population_size = _population_size;
    return status;
}
```

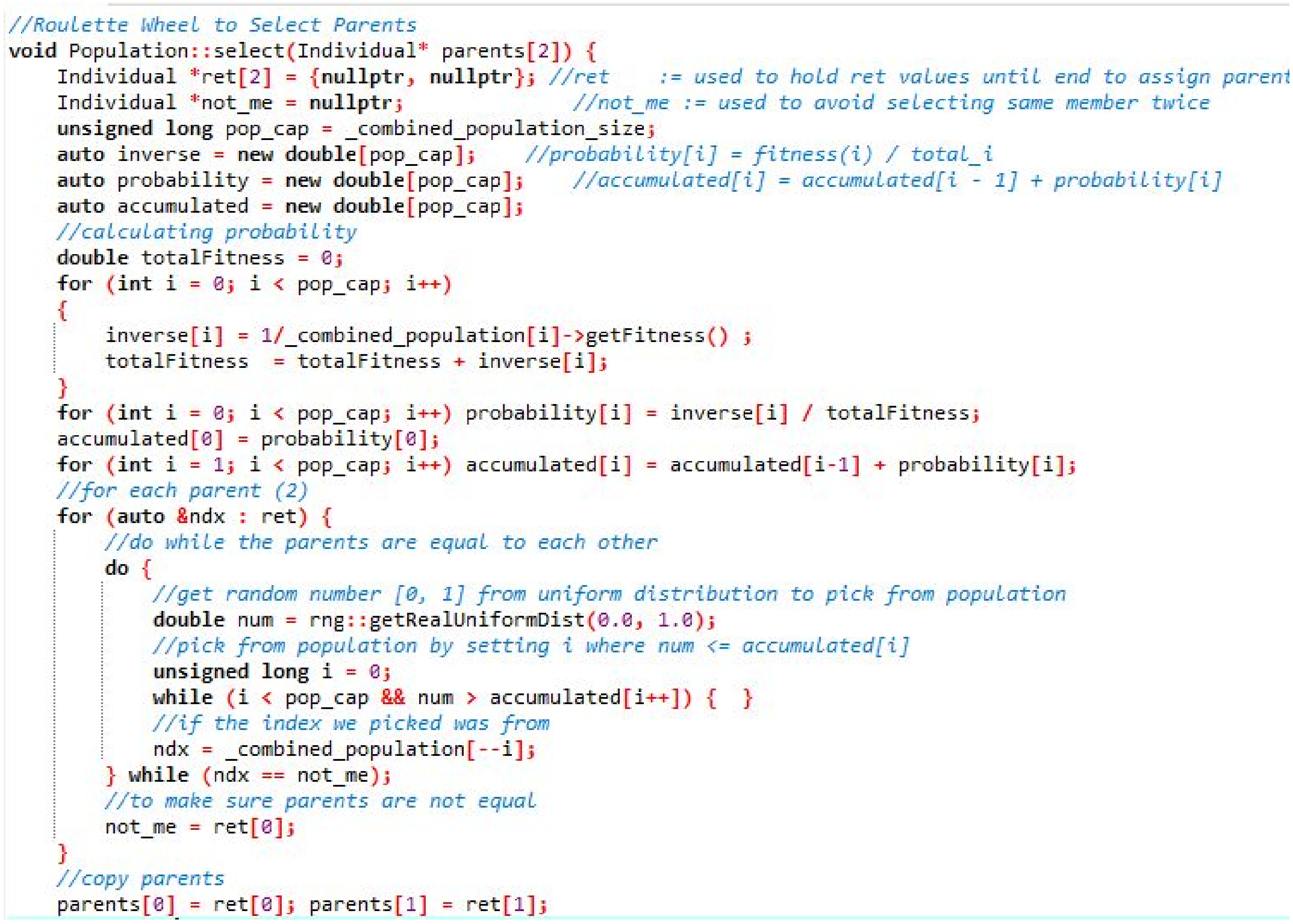
2. **Roulette Wheel**





Chromosome	Fitness Value
A	8.2
B	3.2
C	1.4
D	1.2
E	4.2
F	0.3

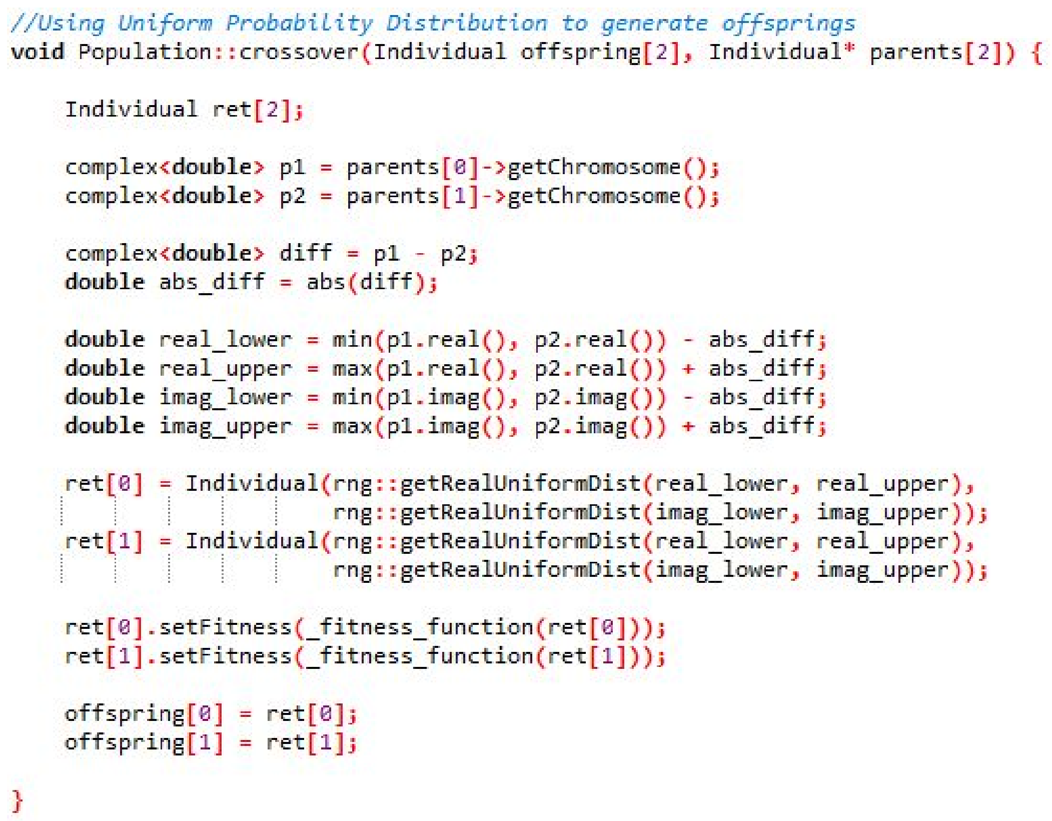
In this method, we do a fitness proportionate selection. This means that individuals which have the highest fitness, have a greater probability to be selected as parents. In our fitness function, since fitness is inversely proportional to the actual fitness of the solution, we create an array to invert the fitness values and then find the accumulated probability of each of these fitness values. We then select a number from [0,1] using uniform probability distribution and choose that individual has a parent whose accumulated probability is just greater than the generated number. Since individuals with higher fitness will span a greater range, they have a higher probability to be selected as parents.



```
//Roulette Wheel to Select Parents
void Population::select(Individual* parents[2]) {
    Individual *ret[2] = {nullptr, nullptr}; //ret    := used to hold ret values until end to assign parent
    Individual *not_me = nullptr;              //not_me := used to avoid selecting same member twice
    unsigned long pop_cap = _combined_population_size;
    auto inverse = new double[pop_cap];    //probability[i] = fitness(i) / total_i
    auto probability = new double[pop_cap];    //accumulated[i] = accumulated[i - 1] + probability[i]
    auto accumulated = new double[pop_cap];
    //calculating probability
    double totalFitness = 0;
    for (int i = 0; i < pop_cap; i++)
    {
        inverse[i] = 1/_combined_population[i]->getFitness() ;
        totalFitness  = totalFitness + inverse[i];
    }
    for (int i = 0; i < pop_cap; i++) probability[i] = inverse[i] / totalFitness;
    accumulated[0] = probability[0];
    for (int i = 1; i < pop_cap; i++) accumulated[i] = accumulated[i-1] + probability[i];
    //for each parent (2)
    for (auto &ndx : ret) {
        //do while the parents are equal to each other
        do {
            //get random number [0, 1] from uniform distribution to pick from population
            double num = rng::getRealUniformDist(0.0, 1.0);
            //pick from population by setting i where num <= accumulated[i]
            unsigned long i = 0;
            while (i < pop_cap && num > accumulated[i++]) {  }
            //if the index we picked was from
            ndx = _combined_population[--i];
        } while (ndx == not_me);
        //to make sure parents are not equal
        not_me = ret[0];
    }
    //copy parents
    parents[0] = ret[0]; parents[1] = ret[1];
```

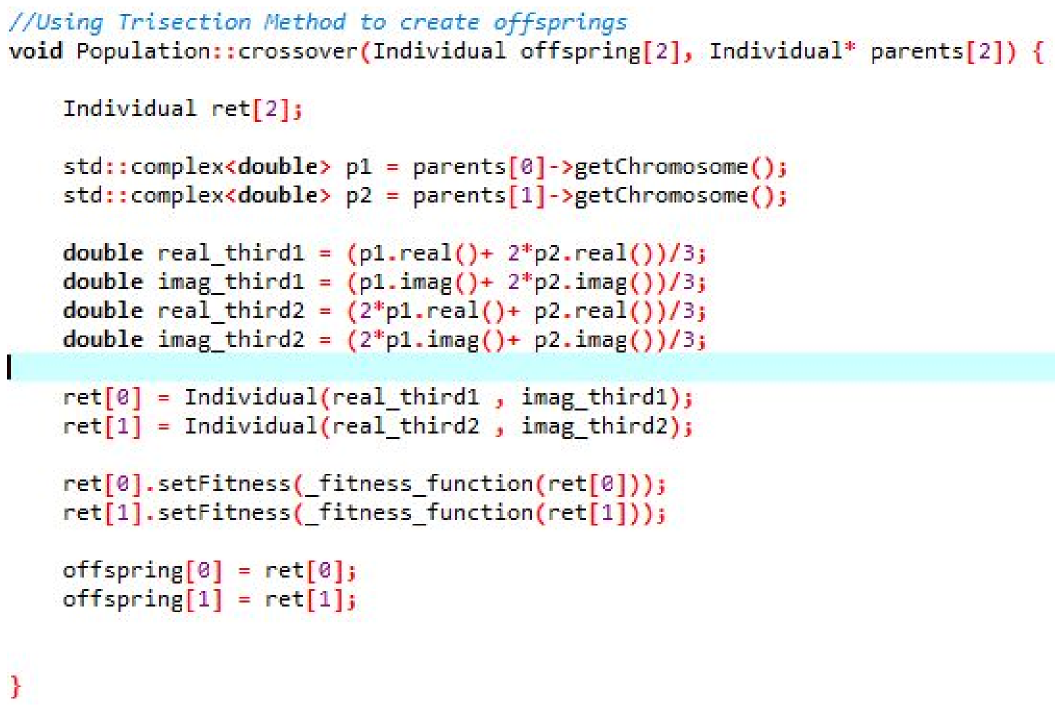
Crossover

1. **Uniform Distribution**
   In this method, we use a uniform probability distribution to generate offsprings by using the two parents. We select a lower bound for the child's both real and imaginary parts by taking the minimum of real and imaginary parts of the parents respectively and subtracting the absolute difference of the values of both parents. Similarly, we select an upper bound for the child's both real and imaginary parts by taking the maximum of real and imaginary parts of the parents respectively and adding the absolute difference of the values of both parents. Finally, we generate two offsprings using uniform probability distribution on the obtained lower and upper bounds for both their real and imaginary parts.



```
//Using Uniform Probability Distribution to generate offsprings
void Population::crossover(Individual offspring[2], Individual* parents[2]) {

    Individual ret[2];

    complex<double> p1 = parents[0]->getChromosome();
    complex<double> p2 = parents[1]->getChromosome();

    complex<double> diff = p1 - p2;
    double abs_diff = abs(diff);

    double real_lower = min(p1.real(), p2.real()) - abs_diff;
    double real_upper = max(p1.real(), p2.real()) + abs_diff;
    double imag_lower = min(p1.imag(), p2.imag()) - abs_diff;
    double imag_upper = max(p1.imag(), p2.imag()) + abs_diff;

    ret[0] = Individual(rng::getRealUniformDist(real_lower, real_upper),
                        rng::getRealUniformDist(imag_lower, imag_upper));
    ret[1] = Individual(rng::getRealUniformDist(real_lower, real_upper),
                        rng::getRealUniformDist(imag_lower, imag_upper));

    ret[0].setFitness(_fitness_function(ret[0]));
    ret[1].setFitness(_fitness_function(ret[1]));

    offspring[0] = ret[0];
    offspring[1] = ret[1];

}
```

2. **Trisection Method**
   Unlike uniform probability distribution, we symmetrically crossover the parents to create two offsprings. The first offspring is created by trisecting the parents in the ratio of 1:2. The second offspring is created by trisecting the parents in the ratio 2:1. Hence, unlike the previous case, the parents mate more symmetrically to produce the offsprings.



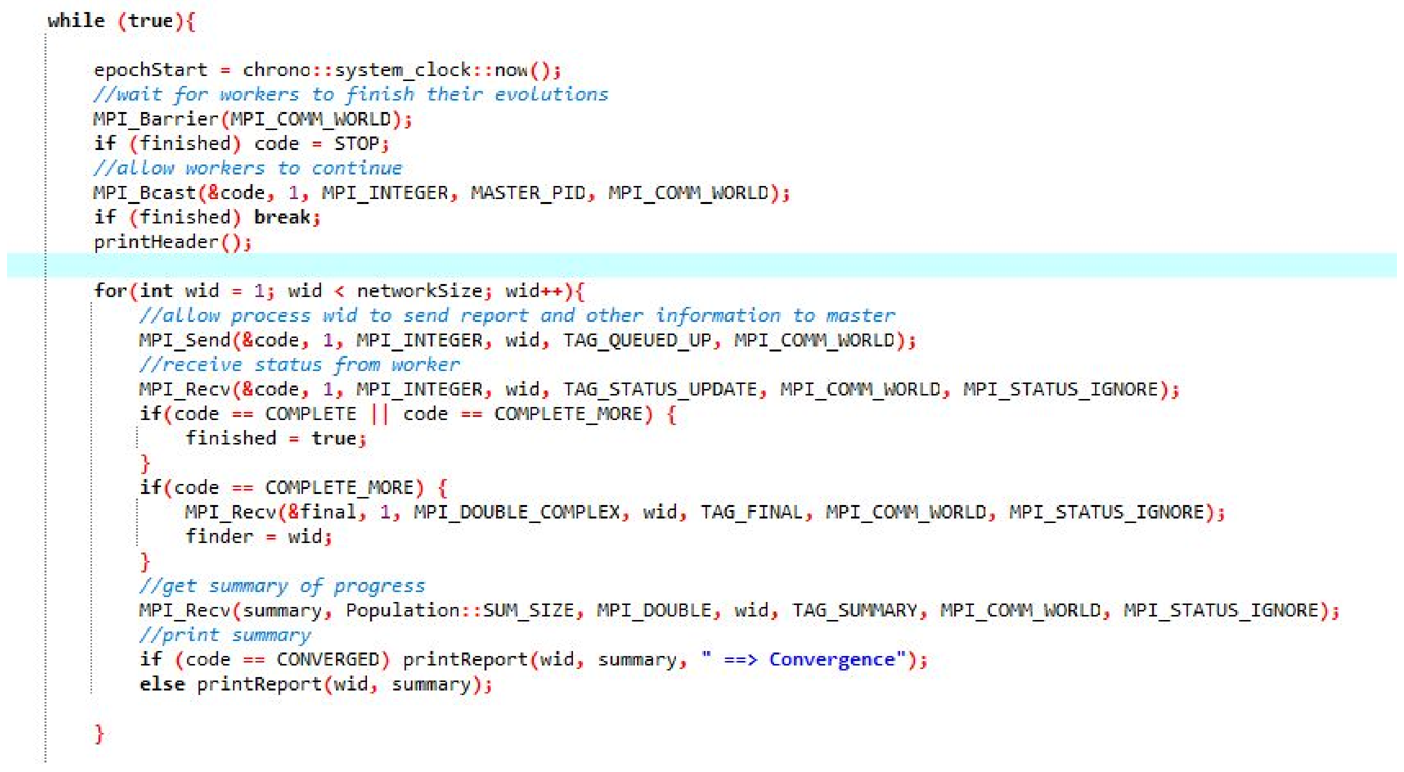
```
//Using Trisection Method to create offsprings
void Population::crossover(Individual offspring[2], Individual* parents[2]) {

    Individual ret[2];

    std::complex<double> p1 = parents[0]->getChromosome();
    std::complex<double> p2 = parents[1]->getChromosome();

    double real_third1 = (p1.real()+ 2*p2.real())/3;
    double imag_third1 = (p1.imag()+ 2*p2.imag())/3;
    double real_third2 = (2*p1.real()+ p2.real())/3;
    double imag_third2 = (2*p1.imag()+ p2.imag())/3;

    ret[0] = Individual(real_third1 , imag_third1);
    ret[1] = Individual(real_third2 , imag_third2);

    ret[0].setFitness(_fitness_function(ret[0]));
    ret[1].setFitness(_fitness_function(ret[1]));

    offspring[0] = ret[0];
    offspring[1] = ret[1];


}
```

MPI Implementation

The MPI implementation of this algorithm involves the worker processes each having its own population independent of other worker processes. All the above-mentioned logic is applied to each of the populations to get varying results of varying accuracies, out of which we select the best result. The master process coordinates all the worker processes.
Whenever a suitable solution is found in any of the processes, the master sees to the termination of all the worker processes printing the best result out of all the worker processes. We make use of multiple tags which allow this mechanism to occur. There are tags for the code completion upon obtaining a solution within the permissible error region, for convergence detection of the population in which case we mutate the entire population to prevent wastage of computation resources to obtain faster results.

The Master Process

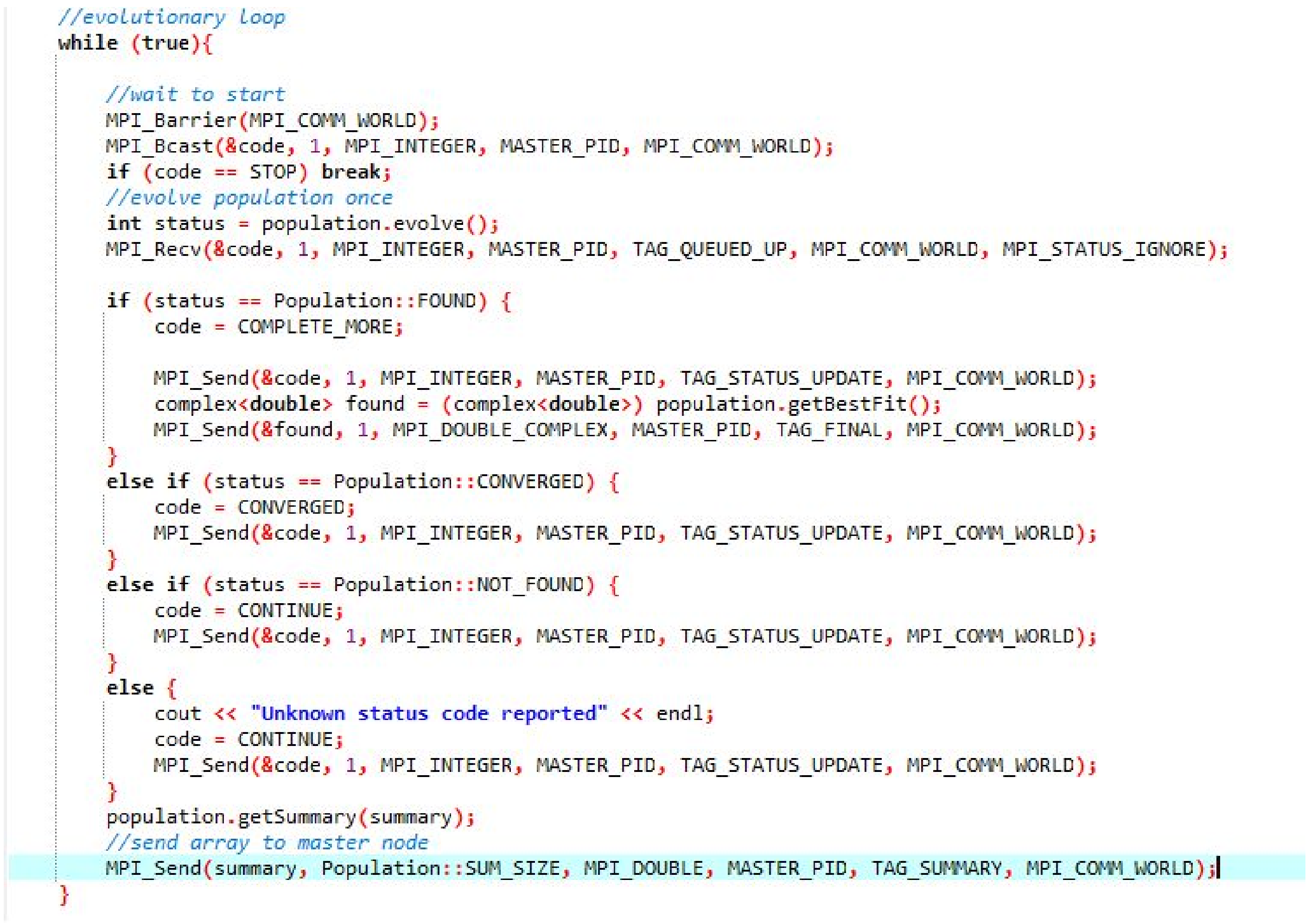
The master process in the below code snippet records the computation time of each generation along with regulating the synchronous working of the workers.
Upon the occurrence of a solution, we find the worker process responsible to find the solution and print the appropriate information.
In addition to printing the best worker population at the end of the algorithm, we print each worker's best results.



```
while (true){

    epochStart = chrono::system_clock::now();
    //wait for workers to finish their evolutions
    MPI_Barrier(MPI_COMM_WORLD);
    if (finished) code = STOP;
    //allow workers to continue
    MPI_Bcast(&code, 1, MPI_INTEGER, MASTER_PID, MPI_COMM_WORLD);
    if (finished) break;
    printHeader();

    for(int wid = 1; wid < networkSize; wid++){
        //allow process wid to send report and other information to master
        MPI_Send(&code, 1, MPI_INTEGER, wid, TAG_QUEUED_UP, MPI_COMM_WORLD);
        //receive status from worker
        MPI_Recv(&code, 1, MPI_INTEGER, wid, TAG_STATUS_UPDATE, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        if(code == COMPLETE || code == COMPLETE_MORE) {
            finished = true;
        }
        if(code == COMPLETE_MORE) {
            MPI_Recv(&final, 1, MPI_DOUBLE_COMPLEX, wid, TAG_FINAL, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            finder = wid;
        }
        //get summary of progress
        MPI_Recv(summary, Population::SUM_SIZE, MPI_DOUBLE, wid, TAG_SUMMARY, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        //print summary
        if (code == CONVERGED) printReport(wid, summary, " ==> Convergence");
        else printReport(wid, summary);

    }
```

The Worker Process

The worker processes (code below) receive statuses at the beginning of each iteration and continue with evolving in case of feasible status. Each worker relays their population's information at the end of each generation. In case of a ‘stop’ status, the workers terminate after sending their best results.

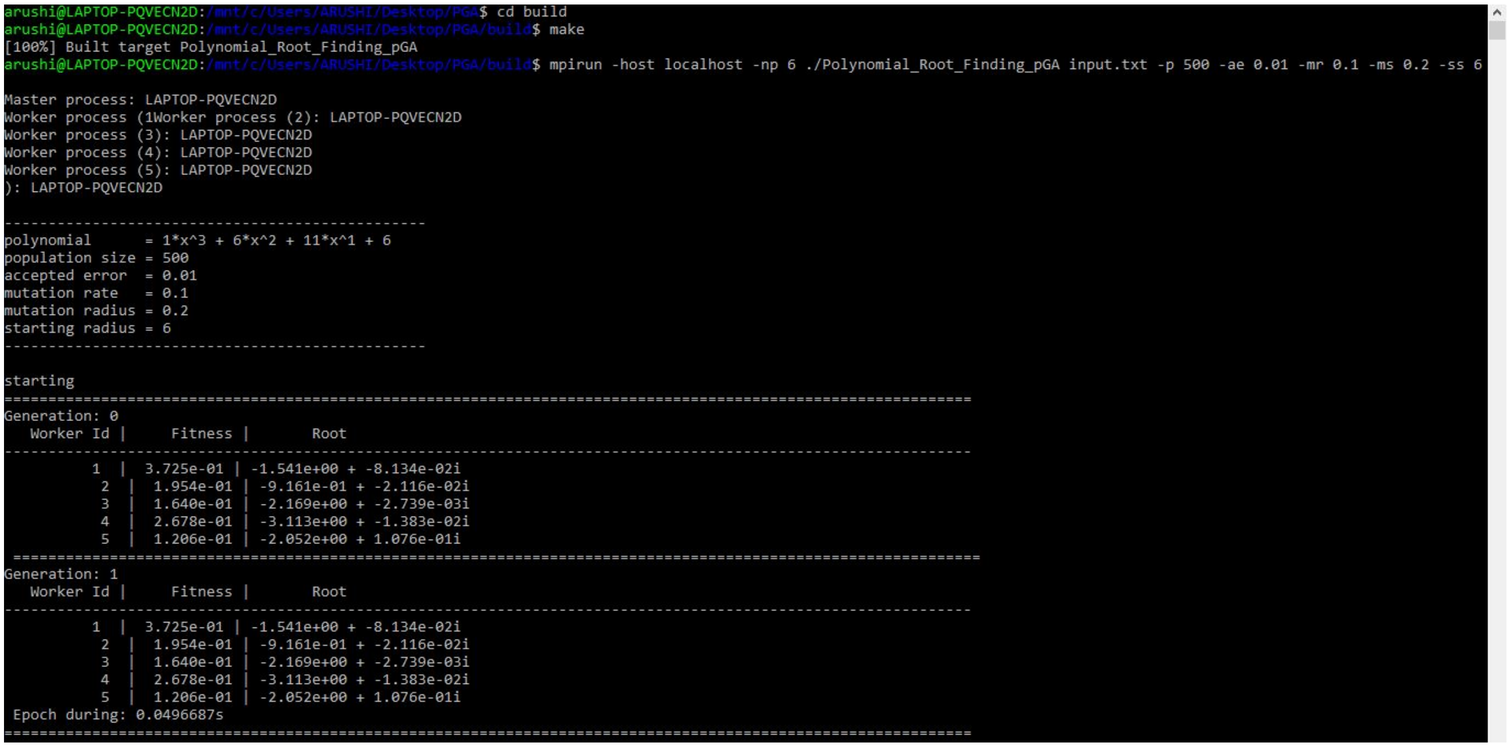


```
//evolutionary loop
while (true){

    //wait to start
    MPI_Barrier(MPI_COMM_WORLD);
    MPI_Bcast(&code, 1, MPI_INTEGER, MASTER_PID, MPI_COMM_WORLD);
    if (code == STOP) break;
    //evolve population once
    int status = population.evolve();
    MPI_Recv(&code, 1, MPI_INTEGER, MASTER_PID, TAG_QUEUED_UP, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

    if (status == Population::FOUND) {
        code = COMPLETE_MORE;

        MPI_Send(&code, 1, MPI_INTEGER, MASTER_PID, TAG_STATUS_UPDATE, MPI_COMM_WORLD);
        complex<double> found = (complex<double>) population.getBestFit();
        MPI_Send(&found, 1, MPI_DOUBLE_COMPLEX, MASTER_PID, TAG_FINAL, MPI_COMM_WORLD);
    }
    else if (status == Population::CONVERGED) {
        code = CONVERGED;
        MPI_Send(&code, 1, MPI_INTEGER, MASTER_PID, TAG_STATUS_UPDATE, MPI_COMM_WORLD);
    }
    else if (status == Population::NOT_FOUND) {
        code = CONTINUE;
        MPI_Send(&code, 1, MPI_INTEGER, MASTER_PID, TAG_STATUS_UPDATE, MPI_COMM_WORLD);
    }
    else {
        cout << "Unknown status code reported" << endl;
        code = CONTINUE;
        MPI_Send(&code, 1, MPI_INTEGER, MASTER_PID, TAG_STATUS_UPDATE, MPI_COMM_WORLD);
    }
    population.getSummary(summary);
    //send array to master node
    MPI_Send(summary, Population::SUM_SIZE, MPI_DOUBLE, MASTER_PID, TAG_SUMMARY, MPI_COMM_WORLD);
}
```

Sample Outputs

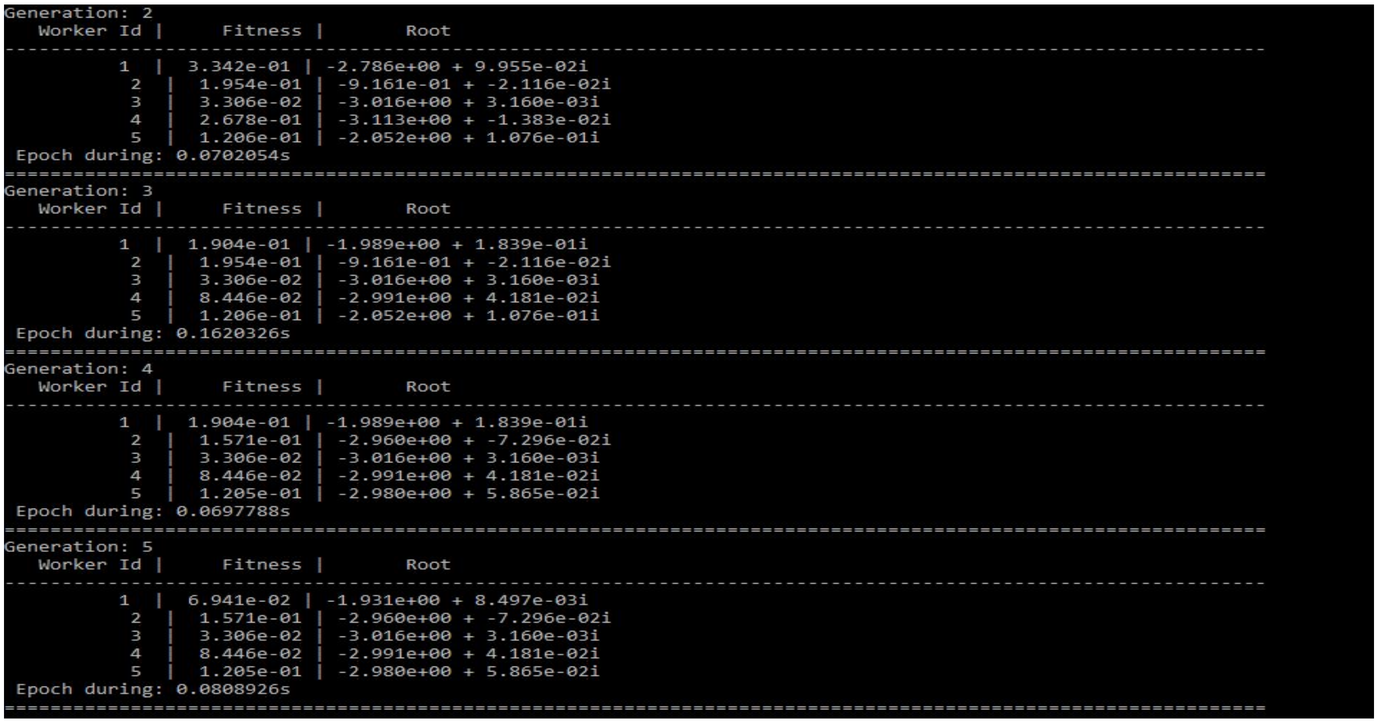
$f(x) = x^3 + 6x^2 + 11x + 6$

Population size	= 500
Accepted error	= 0.01
Mutation rate	= 0.1
Mutation radius	= 0.2
Starting radius	= 6
Number of Parallel Workers	= 5

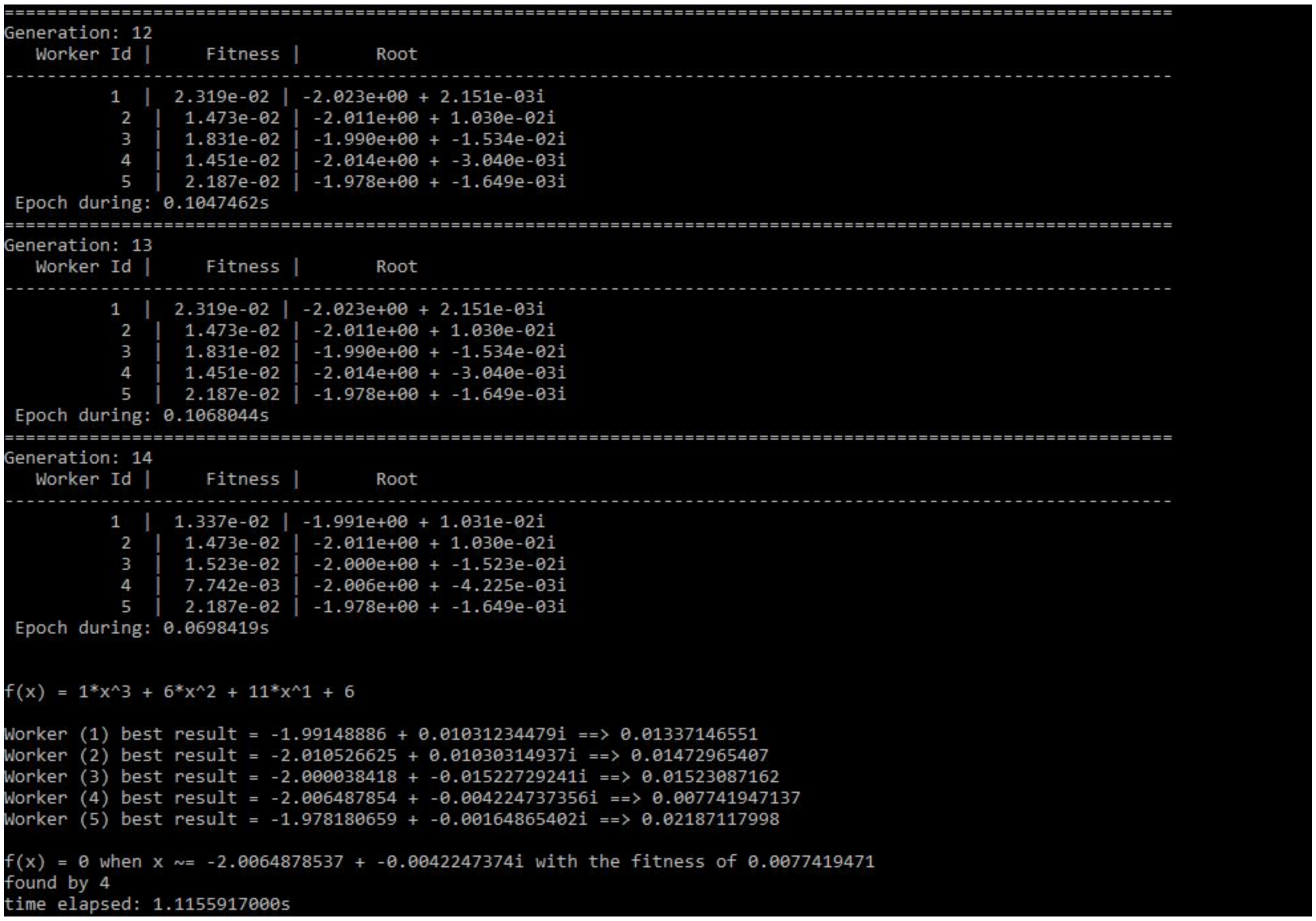


```
arushi@LAPTOP-PQVECN2D:/mnt/c/Users/ARUSHI/Desktop/PGA$ cd build
arushi@LAPTOP-PQVECN2D:/mnt/c/Users/ARUSHI/Desktop/PGA/build$ make
[100%] Built target Polynomial_Root_Finding_pGA
arushi@LAPTOP-PQVECN2D:/mnt/c/Users/ARUSHI/Desktop/PGA/build$ mpirun -host localhost -np 6 ./Polynomial_Root_Finding_pGA input.txt -p 500 -ae 0.01 -mr 0.1 -ms 0.2 -ss 6

Master process: LAPTOP-PQVECN2D
Worker process (1Worker process (2): LAPTOP-PQVECN2D
Worker process (3): LAPTOP-PQVECN2D
Worker process (4): LAPTOP-PQVECN2D
Worker process (5): LAPTOP-PQVECN2D
): LAPTOP-PQVECN2D

------------------------------------------------
polynomial      = 1*x^3 + 6*x^2 + 11*x^1 + 6
population size = 500
accepted error  = 0.01
mutation rate   = 0.1
mutation radius = 0.2
starting radius = 6
------------------------------------------------

starting
==============================================================================================================
Generation: 0
   Worker Id |     Fitness |       Root
--------------------------------------------------------------------------------------------------------------
          1  |  3.725e-01 | -1.541e+00 + -8.134e-02i
           2  |  1.954e-01 | -9.161e-01 + -2.116e-02i
           3  |  1.640e-01 | -2.169e+00 + -2.739e-03i
           4  |  2.678e-01 | -3.113e+00 + -1.383e-02i
           5  |  1.206e-01 | -2.052e+00 + 1.076e-01i
 ==============================================================================================================
Generation: 1
   Worker Id |     Fitness |       Root
--------------------------------------------------------------------------------------------------------------
          1  |  3.725e-01 | -1.541e+00 + -8.134e-02i
           2  |  1.954e-01 | -9.161e-01 + -2.116e-02i
           3  |  1.640e-01 | -2.169e+00 + -2.739e-03i
           4  |  2.678e-01 | -3.113e+00 + -1.383e-02i
           5  |  1.206e-01 | -2.052e+00 + 1.076e-01i
 Epoch during: 0.0496687s
==============================================================================================================
```

We can see the fittest individual generated by each worker during initialization. Each worker continues evolving its population from its 0th generation.
As the generations progress, we see that different workers are converging to different roots (-1, -2, -3) generation after generation.
However, after mutations, most of the workers converge to a single root (-2). The master process terminates when a worker reaches a solution that lies within the accepted error.



```
Generation: 2
   Worker Id |     Fitness |       Root
--------------------------------------------------------------------------------------------------------------
          1  |  3.342e-01 | -2.786e+00 + 9.955e-02i
           2  |  1.954e-01 | -9.161e-01 + -2.116e-02i
           3  |  3.306e-02 | -3.016e+00 + 3.160e-03i
           4  |  2.678e-01 | -3.113e+00 + -1.383e-02i
           5  |  1.206e-01 | -2.052e+00 + 1.076e-01i
 Epoch during: 0.0702054s
==============================================================================================================
Generation: 3
   Worker Id |     Fitness |       Root
--------------------------------------------------------------------------------------------------------------
          1  |  1.904e-01 | -1.989e+00 + 1.839e-01i
           2  |  1.954e-01 | -9.161e-01 + -2.116e-02i
           3  |  3.306e-02 | -3.016e+00 + 3.160e-03i
           4  |  8.446e-02 | -2.991e+00 + 4.181e-02i
           5  |  1.206e-01 | -2.052e+00 + 1.076e-01i
 Epoch during: 0.1620326s
==============================================================================================================
Generation: 4
   Worker Id |     Fitness |       Root
--------------------------------------------------------------------------------------------------------------
          1  |  1.904e-01 | -1.989e+00 + 1.839e-01i
           2  |  1.571e-01 | -2.960e+00 + -7.296e-02i
           3  |  3.306e-02 | -3.016e+00 + 3.160e-03i
           4  |  8.446e-02 | -2.991e+00 + 4.181e-02i
           5  |  1.205e-01 | -2.980e+00 + 5.865e-02i
 Epoch during: 0.0697788s
==============================================================================================================
Generation: 5
   Worker Id |     Fitness |       Root
--------------------------------------------------------------------------------------------------------------
          1  |  6.941e-02 | -1.931e+00 + 8.497e-03i
           2  |  1.571e-01 | -2.960e+00 + -7.296e-02i
           3  |  3.306e-02 | -3.016e+00 + 3.160e-03i
           4  |  8.446e-02 | -2.991e+00 + 4.181e-02i
           5  |  1.205e-01 | -2.980e+00 + 5.865e-02i
 Epoch during: 0.0808926s
==============================================================================================================
```



```
================================================================================================================
Generation: 12
   Worker Id |     Fitness |       Root
----------------------------------------------------------------------------------------------------------------
          1  |  2.319e-02 | -2.023e+00 + 2.151e-03i
           2  |  1.473e-02 | -2.011e+00 + 1.030e-02i
           3  |  1.831e-02 | -1.990e+00 + -1.534e-02i
           4  |  1.451e-02 | -2.014e+00 + -3.040e-03i
           5  |  2.187e-02 | -1.978e+00 + -1.649e-03i
 Epoch during: 0.1047462s
================================================================================================================
Generation: 13
   Worker Id |     Fitness |       Root
----------------------------------------------------------------------------------------------------------------
          1  |  2.319e-02 | -2.023e+00 + 2.151e-03i
           2  |  1.473e-02 | -2.011e+00 + 1.030e-02i
           3  |  1.831e-02 | -1.990e+00 + -1.534e-02i
           4  |  1.451e-02 | -2.014e+00 + -3.040e-03i
           5  |  2.187e-02 | -1.978e+00 + -1.649e-03i
 Epoch during: 0.1068044s
================================================================================================================
Generation: 14
   Worker Id |     Fitness |       Root
----------------------------------------------------------------------------------------------------------------
          1  |  1.337e-02 | -1.991e+00 + 1.031e-02i
           2  |  1.473e-02 | -2.011e+00 + 1.030e-02i
           3  |  1.523e-02 | -2.000e+00 + -1.523e-02i
           4  |  7.742e-03 | -2.006e+00 + -4.225e-03i
           5  |  2.187e-02 | -1.978e+00 + -1.649e-03i
 Epoch during: 0.0698419s


f(x) = 1*x^3 + 6*x^2 + 11*x^1 + 6

Worker (1) best result = -1.99148886 + 0.01031234479i ==> 0.01337146551
Worker (2) best result = -2.010526625 + 0.01030314937i ==> 0.01472965407
Worker (3) best result = -2.000038418 + -0.01522729241i ==> 0.01523087162
Worker (4) best result = -2.006487854 + -0.004224737356i ==> 0.007741947137
Worker (5) best result = -1.978180659 + -0.00164865402i ==> 0.02187117998

f(x) = 0 when x ~= -2.0064878537 + -0.0042247374i with the fitness of 0.0077419471
found by 4
time elapsed: 1.1155917000s
```

Results Table and Comparison

1. Using different selection and evolution methods

$f(x) = (2 + 1i)x^5 + 6x^4 + 7ix^3 + x^2 + (9 + 4i)x + 2$

Population size	= 500
Accepted error	= 0.01
Mutation rate	= 0.1
Mutation radius	= 0.2
Starting radius	= 6
Number of Parallel Workers	= 5

Selection	Evolution	Generations	Time (s)	Best Result	Fitness
Elitism	Uniform Distribution	26	1.778	-0.1795+0.08i	0.0099
Elitism	Trisection Method	10	0.691	-0.1806+0.0861i	0.0014
Roulette Wheel	Uniform Distribution	5	0.651	-0.1804+0.0870i	0.0089
Roulette Wheel	Trisection Method	4	0.471	-0.1800+0.0863i	0.0053

Inference

We find that Elitism combined with Uniform Distribution takes the maximum time and generations for calculation while Roulette Wheel combined with Trisection Method takes minimum time and generation. This is mainly because the trisection method is much more symmetric than uniform probability distribution and elitism is much more biased towards fitter population compared to Roulette Wheel, thus sticking to a select number of fit parents. Since we always get the elitist of the population reproducing symmetrically, fitness is maximized and we reach solutions faster.

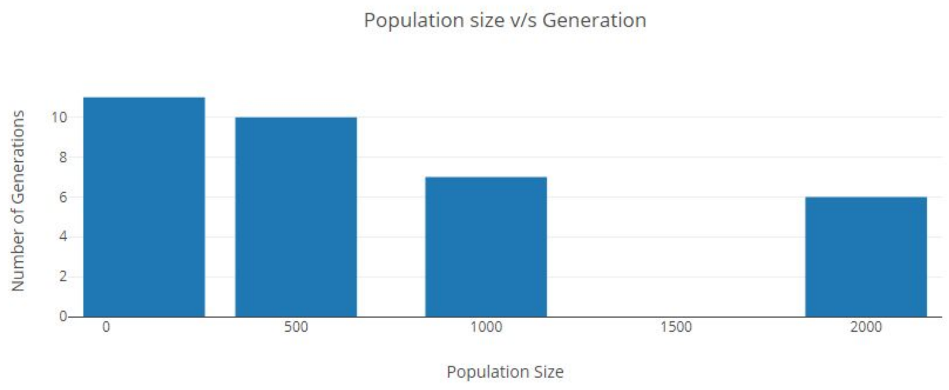
2. Varying Population Size

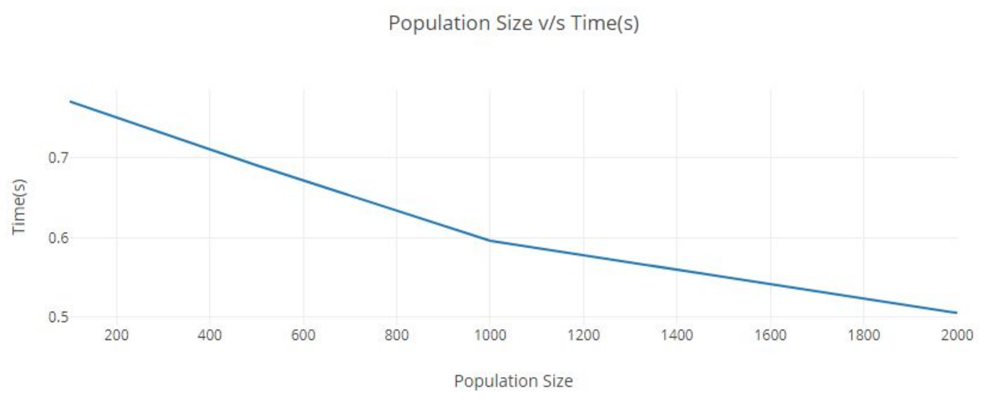
$f(x) = (2 + 1i)x^5 + 6x^4 + 7ix^3 + x^2 + (9 + 4i)x + 2$

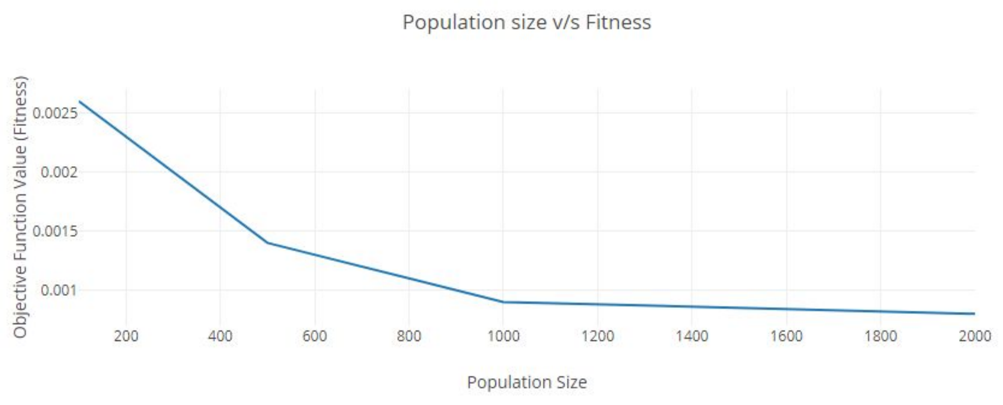
Accepted error	= 0.01
Mutation rate	= 0.1
Mutation radius	= 0.2
Starting radius	= 6

Selection : Elitism
Evolution : Trisection
Number of Workers: 5

Population	**Generations**	**Time (s)**	**Best Result**	**Fitness**
100	11	0.771	-0.1804+0.0863i	0.0026
500	10	0.691	-0.1806+0.0861i	0.0014
1000	7	0.596	-0.1813+0.0864i	0.0009
2000	6	0.5055	-0.1808+0.0869i	0.0008







Inference

We see that as the population size increases, our algorithm gets more refined (high fitness solutions) and we can find solutions much faster with less number of generations. This is because more the population size, greater the span over a range of solutions and hence greater probability of getting fitter solutions. However, we need to be careful with increasing population since it can lead to the overhead of increased computations.

3. Comparison with the Sequential Approach (No master, only a single thread running)

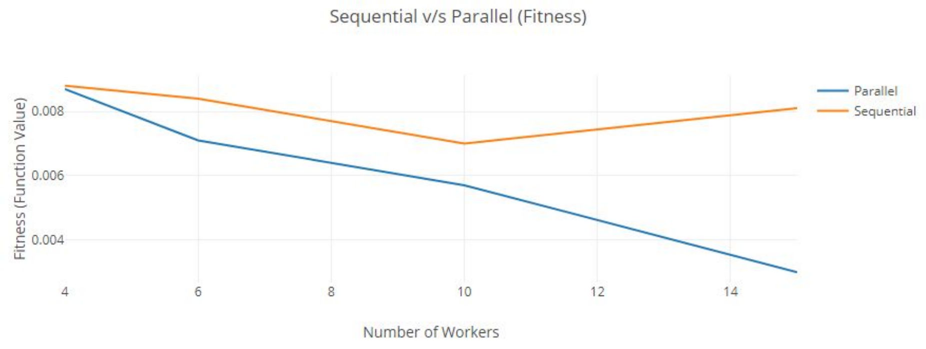
$f(x) = (2 + 1i)x^5 + 6x^4 + 7ix^3 + x^2 + (9 + 4i)x + 2$

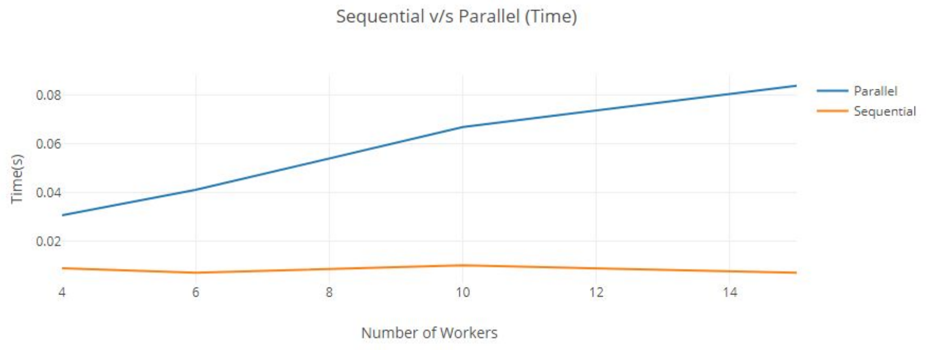
Accepted error	= 0.01
Mutation rate	= 0.1
Mutation radius	= 0.2
Starting radius	= 6

Selection : Elitism, Evolution : Trisection

	Workers	Time (s)	Best Result	Fitness
Sequential	-	0.0088	-0.1797 + 0.0857i	0.0088
Parallel	4	0.0307	-0.1800 + 0.0856i	0.0087
Sequential	-	0.0070	-0.1805 + 0.0864i	0.0084
Parallel	6	0.0411	-0.1799 + 0.0857i	0.0071

	Workers	Time (s)	Best Result	Fitness
Sequential	-	0.0100	-0.1804 + 0.0861i	0.0070
Parallel	10	0.0670	-0.1804 + 0.0869i	0.0057
Sequential	-	0.0070	-0.1809 + 0.0856i	0.0081
Parallel	15	0.0840	-0.1802 + 0.0860i	0.0030





Inference

- ➢ We see the accuracy of our parallel algorithm increasing i.e. the fitness of the best result decrease roughly with the increase in the number of workers whereas the sequential fitness is random since the convergence of the population on a particular solution is a random event based on the evolution methods.

- ➢ The time required for finding the solution in case of the parallel algorithm increases since the overhead for managing more processes is substantial due to communication between masters and slaves.

The purpose of having multiple workers is to arrive at the solution faster than the sequential algorithm since the convergence on a solution is an event based on the probability of the initially assigned individuals and the random selection of parents. The more the workers, our chance of reaching a more accurate solution faster than a single thread algorithm increases. But the communication overhead comes into action and the time adds up to be more than that of the sequential algorithm.

However, due to the multiple workers, we get a fitter result each time the number of workers increase due to the probability factor mentioned above. Each worker proceeds to a random point of convergence giving us a better chance at the solution.

Thus, the parallel approach triumphs over the sequential one in terms of accuracy but fails in case of time.

Conclusion and Future Scope

We have developed a parallel genetic algorithm using MPI Master Slave Method to find roots of a complex polynomial equation. We have used various parent selection and crossover techniques to find the best evolution method that increases fitness within lesser time frames and generations. We used multiple workers to solve the same equation so as to increase the probability of reaching a solution faster as well as more accurately. We compared our results by taking different combinations of evolution methods, a different population size and the parallel approach variable workers to its sequential counterpart where no master or slaves are involved.

We can extend this project to find all the roots of the given complex polynomial. This can be achieved if each worker finds a distinct root for the equation by using an appropriate fitness function that reduces the fitness infinitely when a worker solution hovers around a root that is already reached.

References

https://www.mcs.anl.gov/~itf/dbpp/text/node96.html

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

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

https://pdfs.semanticscholar.org/ff16/8d2d0bb304c1aaf1b5f1a48aa5012e55c7a8.pdf

https://pdfs.semanticscholar.org/a7ea/ea18b96328d8b5ffba70f05e44f679e95a46.pdf

https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4365503

Instructions for running the code

1. Go to build directory
2. Populate the input file in the following format:
   Degree of Polynomial (say n)
   Real Imaginary (for next n lines)
3. Run the following commands
   a. make
   b. mpirun -host localhost -np 6 ./Polynomial_Root_Finding_pGA input.txt -p 500 -ae 0.01 -mr 0.1 -ms 0.2 -ss 6 (for PGA)
      ./Polynomial_Root_Finding_pGA input.txt -p 500 -ae 0.01 -mr 0.1 -ms 0.2 -ss 6 (for SGA)

Notes regarding the code

1. The code has been appropriately commented everywhere and has extensively used classes
2. The commented code in population.cpp is for various combinations of parent selection and crossover (Snippets in the report)
3. The following important in-built functions from **MPI library** have been used:

a.	MPI_INIT	Initiate an MPI computation
b.	MPI_COMM_SIZE	Determine number of processes
c.	MPI_COMM_RANK	Determine the process identifier.
d.	MPI_SEND	Send a message
e.	MPI_RECV	Receive a message
f.	MPI_FINALIZE	Terminate a computation
g.	MPI_Bcast	Broadcast message