CS 5500

Spring 2021
Homework 8
By Griffin Hackley
A02224681

My implementation works by using a leader-worker configuration. First the leader creates an initial generation by randomly shuffling an array of numbers from 1-100. Then it goes into the while loop of the algorithm. The first thing that happens is a breeding pool is created. This is done by sorting the generation from most fit to least fit, and then taking the most fit half of the generation.

After this, the leader will send each worker 2 different lists taken from the breeding pool. Each worker will take these 2 lists and create a child list using PMX. The workers then send the child list back to the leader. The leader then finds the average fitness for the generation and outputs that to the console.

If the average of the last generation and the current generation are the same then the generation mutates. This is to keep the pool from becoming stagnate and keep it improving. Mutation is done by taking the most fit list in the generation, splitting it into 3 parts at random segments, then reversing one of those segments randomly. This new list is then added to the generation. It does not replace any list that is already in that generation. This is done multiple times to get a few different mutations in the generation. The algorithm then restarts by finding the new breeding pool.

The command to compile the code is:

```
mpic++ main.cpp
```

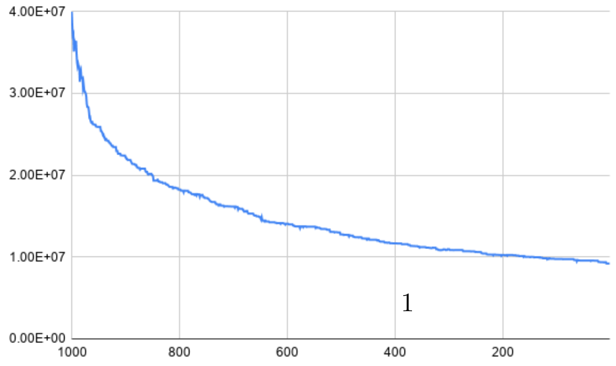
The command to run the code is:

```
mpirun -np (num) -oversubscribe a.out
```

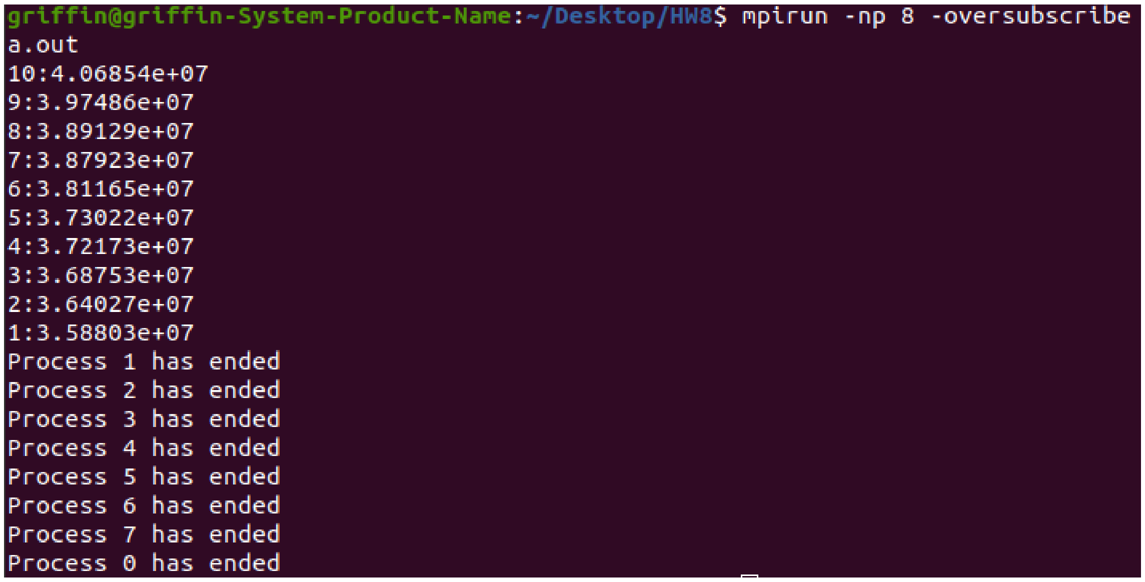
where (num) is the number of processes

Below is a graph of the number of generations left to run vs. the best solution of that generation

The best solution computed was $9.21 * 10^6$ or 9,210,000

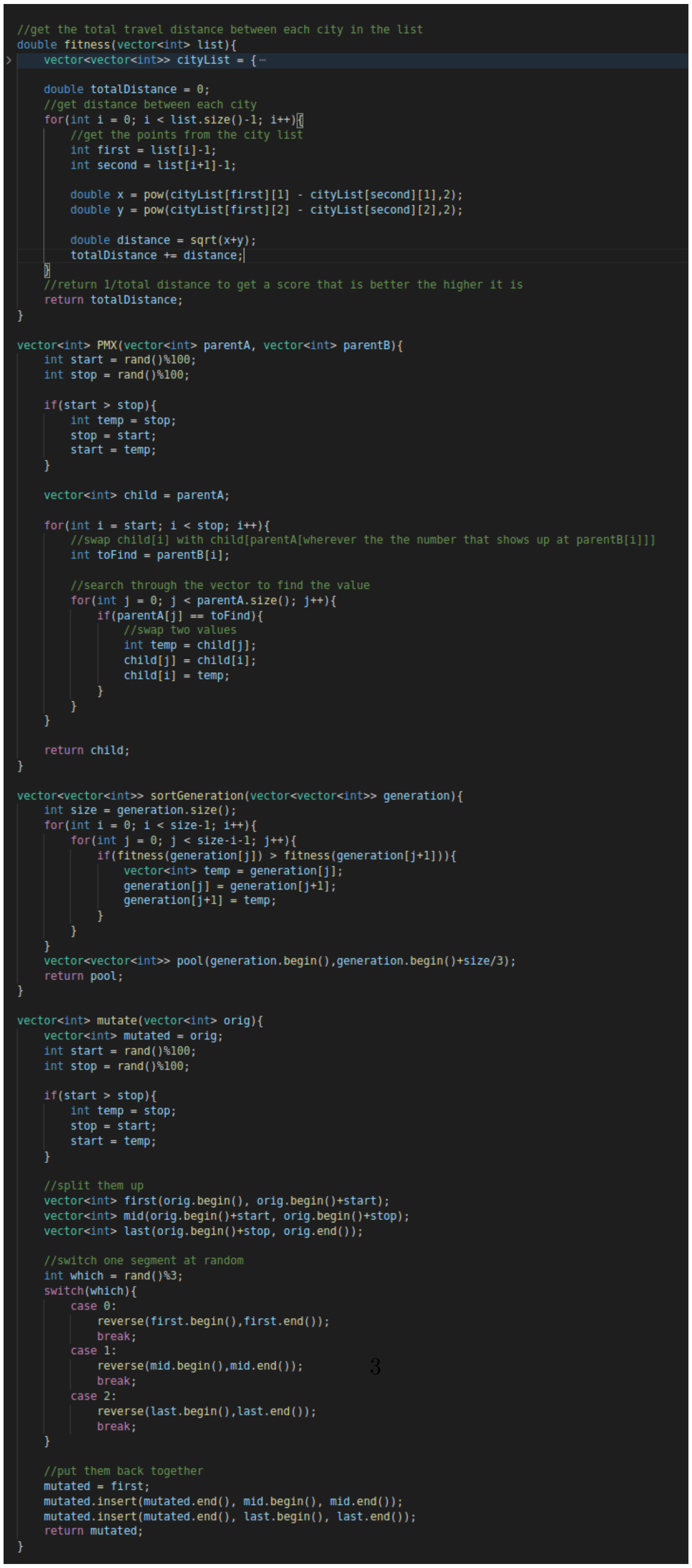


Below are the sample results

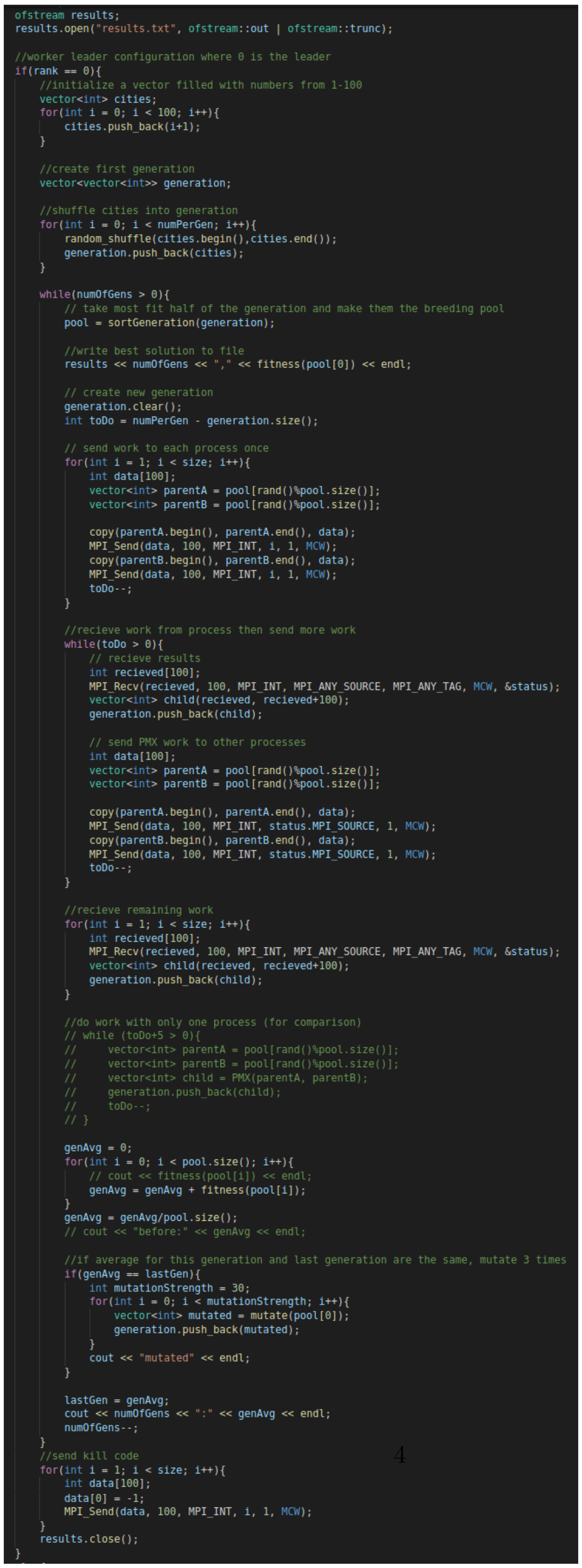


```
griffin@griffin-System-Product-Name:~/Desktop/HW8$ mpirun -np 8 -oversubscribe
a.out
10:4.06854e+07
9:3.97486e+07
8:3.89129e+07
7:3.87923e+07
6:3.81165e+07
5:3.73022e+07
4:3.72173e+07
3:3.68753e+07
2:3.64027e+07
1:3.58803e+07
Process 1 has ended
Process 2 has ended
Process 3 has ended
Process 4 has ended
Process 5 has ended
Process 6 has ended
Process 7 has ended
Process 0 has ended
```

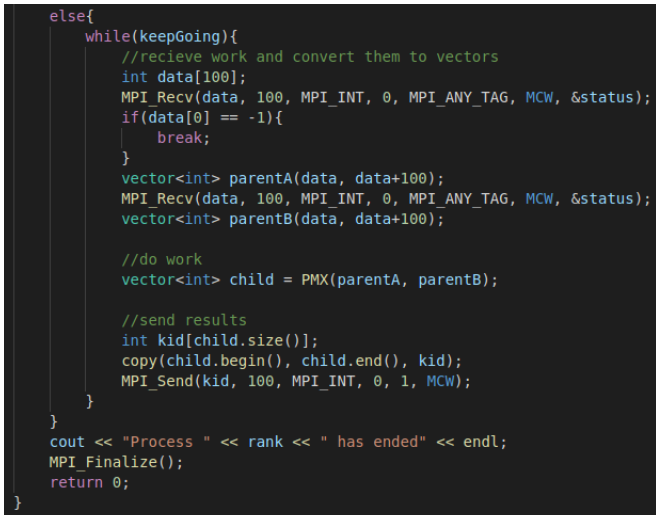
Full code is on the following pages



```cpp
//get the total travel distance between each city in the list
double fitness(vector<int> list){
    vector<vector<int>> cityList = {⋯

    double totalDistance = 0;
    //get distance between each city
    for(int i = 0; i < list.size()-1; i++){
        //get the points from the city list
        int first = list[i]-1;
        int second = list[i+1]-1;

        double x = pow(cityList[first][1] - cityList[second][1],2);
        double y = pow(cityList[first][2] - cityList[second][2],2);

        double distance = sqrt(x+y);
        totalDistance += distance;
    }
    //return 1/total distance to get a score that is better the higher it is
    return totalDistance;
}

vector<int> PMX(vector<int> parentA, vector<int> parentB){
    int start = rand()%100;
    int stop = rand()%100;

    if(start > stop){
        int temp = stop;
        stop = start;
        start = temp;
    }

    vector<int> child = parentA;

    for(int i = start; i < stop; i++){
        //swap child[i] with child[parentA[wherever the the number that shows up at parentB[i]]]
        int toFind = parentB[i];

        //search through the vector to find the value
        for(int j = 0; j < parentA.size(); j++){
            if(parentA[j] == toFind){
                //swap two values
                int temp = child[j];
                child[j] = child[i];
                child[i] = temp;
            }
        }
    }

    return child;
}

vector<vector<int>> sortGeneration(vector<vector<int>> generation){
    int size = generation.size();
    for(int i = 0; i < size-1; i++){
        for(int j = 0; j < size-i-1; j++){
            if(fitness(generation[j]) > fitness(generation[j+1])){
                vector<int> temp = generation[j];
                generation[j] = generation[j+1];
                generation[j+1] = temp;
            }
        }
    }
    vector<vector<int>> pool(generation.begin(),generation.begin()+size/3);
    return pool;
}

vector<int> mutate(vector<int> orig){
    vector<int> mutated = orig;
    int start = rand()%100;
    int stop = rand()%100;

    if(start > stop){
        int temp = stop;
        stop = start;
        start = temp;
    }

    //split them up
    vector<int> first(orig.begin(), orig.begin()+start);
    vector<int> mid(orig.begin()+start, orig.begin()+stop);
    vector<int> last(orig.begin()+stop, orig.end());

    //switch one segment at random
    int which = rand()%3;
    switch(which){
        case 0:
            reverse(first.begin(),first.end());
            break;
        case 1:
            reverse(mid.begin(),mid.end());
            break;
        case 2:
            reverse(last.begin(),last.end());
            break;
    }

    //put them back together
    mutated = first;
    mutated.insert(mutated.end(), mid.begin(), mid.end());
    mutated.insert(mutated.end(), last.begin(), last.end());
    return mutated;
}
```



```cpp
ofstream results;
results.open("results.txt", ofstream::out | ofstream::trunc);

//worker leader configuration where 0 is the leader
if(rank == 0){
    //initialize a vector filled with numbers from 1-100
    vector<int> cities;
    for(int i = 0; i < 100; i++){
        cities.push_back(i+1);
    }

    //create first generation
    vector<vector<int>> generation;

    //shuffle cities into generation
    for(int i = 0; i < numPerGen; i++){
        random_shuffle(cities.begin(),cities.end());
        generation.push_back(cities);
    }

    while(numOfGens > 0){
        // take most fit half of the generation and make them the breeding pool
        pool = sortGeneration(generation);

        //write best solution to file
        results << numOfGens << "," << fitness(pool[0]) << endl;

        // create new generation
        generation.clear();
        int toDo = numPerGen - generation.size();

        // send work to each process once
        for(int i = 1; i < size; i++){
            int data[100];
            vector<int> parentA = pool[rand()%pool.size()];
            vector<int> parentB = pool[rand()%pool.size()];

            copy(parentA.begin(), parentA.end(), data);
            MPI_Send(data, 100, MPI_INT, i, 1, MCW);
            copy(parentB.begin(), parentB.end(), data);
            MPI_Send(data, 100, MPI_INT, i, 1, MCW);
            toDo--;
        }

        //recieve work from process then send more work
        while(toDo > 0){
            // recieve results
            int recieved[100];
            MPI_Recv(recieved, 100, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG, MCW, &status);
            vector<int> child(recieved, recieved+100);
            generation.push_back(child);

            // send PMX work to other processes
            int data[100];
            vector<int> parentA = pool[rand()%pool.size()];
            vector<int> parentB = pool[rand()%pool.size()];

            copy(parentA.begin(), parentA.end(), data);
            MPI_Send(data, 100, MPI_INT, status.MPI_SOURCE, 1, MCW);
            copy(parentB.begin(), parentB.end(), data);
            MPI_Send(data, 100, MPI_INT, status.MPI_SOURCE, 1, MCW);
            toDo--;
        }

        //recieve remaining work
        for(int i = 1; i < size; i++){
            int recieved[100];
            MPI_Recv(recieved, 100, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG, MCW, &status);
            vector<int> child(recieved, recieved+100);
            generation.push_back(child);
        }

        //do work with only one process (for comparison)
        // while (toDo+5 > 0){
        //     vector<int> parentA = pool[rand()%pool.size()];
        //     vector<int> parentB = pool[rand()%pool.size()];
        //     vector<int> child = PMX(parentA, parentB);
        //     generation.push_back(child);
        //     toDo--;
        // }

        genAvg = 0;
        for(int i = 0; i < pool.size(); i++){
            // cout << fitness(pool[i]) << endl;
            genAvg = genAvg + fitness(pool[i]);
        }
        genAvg = genAvg/pool.size();
        // cout << "before:" << genAvg << endl;

        //if average for this generation and last generation are the same, mutate 3 times
        if(genAvg == lastGen){
            int mutationStrength = 30;
            for(int i = 0; i < mutationStrength; i++){
                vector<int> mutated = mutate(pool[0]);
                generation.push_back(mutated);
            }
            cout << "mutated" << endl;
        }

        lastGen = genAvg;
        cout << numOfGens << ":" << genAvg << endl;
        numOfGens--;
    }
    //send kill code
    for(int i = 1; i < size; i++){
        int data[100];
        data[0] = -1;
        MPI_Send(data, 100, MPI_INT, i, 1, MCW);
    }
    results.close();
}
```



```
    else{
        while(keepGoing){
            //recieve work and convert them to vectors
            int data[100];
            MPI_Recv(data, 100, MPI_INT, 0, MPI_ANY_TAG, MCW, &status);
            if(data[0] == -1){
                break;
            }
            vector<int> parentA(data, data+100);
            MPI_Recv(data, 100, MPI_INT, 0, MPI_ANY_TAG, MCW, &status);
            vector<int> parentB(data, data+100);

            //do work
            vector<int> child = PMX(parentA, parentB);

            //send results
            int kid[child.size()];
            copy(child.begin(), child.end(), kid);
            MPI_Send(kid, 100, MPI_INT, 0, 1, MCW);
        }
    }
    cout << "Process " << rank << " has ended" << endl;
    MPI_Finalize();
    return 0;
}
```