<u>**DSAI 3202 – Assignment 2 (using AWS)**</u>

<u>Part 1: Game of Life MPI4PY (40 points)</u>

1. Program files:

functions_game_life.py: includes two main execution functions update_Grid_GameofLife and exchange_ghost_rows.

- <u>**update_Grid_GameofLife**</u>: It updates the local grid based on the Game rules. Taking into account the top and bottom ghost rows for communication with neighboring processes. It consist of another function called count_neighbors, which calculates the total alive neighbours from all 8 directions of each cell. The game rules also take place within this function. It solved the game in O(1) by updating the cell in place.

```python
from mpi4py import MPI
import numpy as np

def update_Grid_GameofLife(local_grid, top_ghost_row, bottom_ghost_row):

    """
        Purpose: Updating local grid according to the rules of Game of Life,
        using the provided top and bottom ghost rows to create an extended grid.

        Parameters (numpy.ndarray):
        - local_grid: local grid to be updated.
        - top_ghost_row: ghost row above the local grid.
        - bottom_ghost_row : ghost row below the local grid.

        Returns:
        - None: The function modifies the `local_grid` array in place (o(1)).

    """

    # Extended grid includes the local grid plus the top and bottom ghost rows
    extended_grid = np.vstack([top_ghost_row, local_grid, bottom_ghost_row])
    n_rows, n_cols = extended_grid.shape

    # Function to count neighbors for extended grid with ghost rows
    def count_neighbors(r, c):
        nei = 0
        for i in range(max(0, r - 1), min(r + 2, n_rows)):
            for j in range(max(0, c - 1), min(c + 2, n_cols)):
                if i == r and j == c:
                    continue
                # checking for live cells
                if extended_grid[i][j] in [1, 3]:
                    nei += 1
        return nei
```

```python

    # Apply Game of Life rules
    for r in range(1, n_rows - 1):  # Exclude ghost rows in iteration
        for c in range(n_cols):
            nei_alive = count_neighbors(r, c)
            if extended_grid[r][c] == 1 and nei_alive in [2, 3]:
                local_grid[r - 1][c] = 3  # Alive to Alive
            elif extended_grid[r][c] == 0 and nei_alive == 3:
                local_grid[r - 1][c] = 2  # Dead to Alive

    # apply the temporary state transitions
    for r in range(local_grid.shape[0]):
        for c in range(local_grid.shape[1]):
            if local_grid[r][c] == 3:
                local_grid[r][c] = 1  # Remain alive
            elif local_grid[r][c] == 2:
                local_grid[r][c] = 1  # Become alive
            else:
                local_grid[r][c] = 0  # Become dead or remain dead

```

- **game_life.py:** includes the script to run the game of life using MPI4y

```python
# game_life.py

from mpi4py import MPI
import numpy as np
from functions_game_life import update_Grid_GameofLife, exchange_ghost_rows

# 1. Initialize MPI
comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()

# 2. Define the grid
grid_rows = 20
grid_cols = 20

# Number of rows per process
rows_per_process = grid_rows // size
remaining_rows = grid_rows % size
if rank < remaining_rows:
    start_row = (rows_per_process + 1) * rank
    local_grid_rows = rows_per_process + 1
else:
    start_row = (rows_per_process + 1) * remaining_rows + rows_per_process * (rank - remaining_rows)
    local_grid_rows = rows_per_process

# 3. Initialize local grid (strip) for each process
local_grid = np.random.randint(2, size=(local_grid_rows, grid_cols))

# Number of steps to simulate
num_steps = 10
```

```
# 5. Simulation loop
for step in range(num_steps):

    # Exchange ghost rows with neighboring processes
    top_ghost_row, bottom_ghost_row = exchange_ghost_rows(local_grid, grid_cols, rank, size, comm)

    # Update the local grid with the new state
    update_Grid_GameofLife(local_grid, top_ghost_row, bottom_ghost_row)

    # Gather the updated grids on the root process
    if rank == 0:
        # container to receive the updated grids from all processes
        full_grid = np.empty((grid_rows, grid_cols), dtype=int)

        # This particularly for large grid size since the console has limitation of number of charc
        print(full_grid.shape)
    else:
        full_grid = None

    # Gather all local grids into the full_grid array on the root process
    comm.Gather(local_grid, full_grid, root=0)

    # 6. Visualization on the root process
    if rank == 0:
        print(f"Step {step}:")
        for row in full_grid:
            print(' '.join(str(cell) for cell in row))
        print("\n" + "="*40 + "\n")
```

2. **Test the script by running it with multiple MPI processes using mpirun command.**

   It ouputs 10 fixed number of simulation steps

```
● (base) ubuntu@ip-172-31-60-236:~$ mpirun -np 4 python3 game_life.py
(20, 20)
Step 0:
0 1 0 0 0 1 1 1 1 1 1 1 1 1 0 1 1 0 1 1
1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
0 0 0 0 0 0 1 0 1 1 0 0 0 0 0 0 0 0 0 1
0 0 0 0 1 0 0 1 1 1 0 0 0 0 0 0 0 0 0 1
0 1 0 0 0 0 0 1 0 1 0 0 0 0 0 1 0 0 0 0
1 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 1 0 1
1 0 0 1 0 1 0 0 0 0 0 0 0 1 0 1 0 0 0 1
0 0 0 0 0 1 0 1 0 0 0 0 0 0 0 0 0 0 1 1
0 0 0 1 0 1 0 0 1 1 1 0 1 0 0 0 0 1 0 1
0 0 0 0 0 1 1 1 0 0 1 0 0 0 0 0 0 1 1 1
1 0 0 0 1 1 1 1 0 1 0 0 0 1 0 0 1 0 0 1
0 1 1 0 1 0 0 0 0 0 0 0 1 0 1 0 0 1 1 0
0 0 0 1 0 0 0 0 1 1 0 1 1 1 1 1 0 0 0 0
0 0 0 1 1 0 0 1 0 0 0 0 0 1 1 0 0 1 0 0
1 1 0 1 1 1 1 1 1 1 0 1 0 0 1 0 0 1 1 0
1 1 0 0 0 0 1 0 0 0 0 1 0 0 1 0 0 0 1 0
0 1 0 1 0 0 1 1 1 1 1 0 0 0 1 0 0 0 1 0
0 1 0 1 0 0 0 0 0 0 0 0 1 1 1 1 0 0 0 0
0 1 1 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 1 1
0 1 1 1 1 0 1 1 0 1 0 0 1 1 0 1 1 1 1 0

========================================

(20, 20)
Step 1:
0 0 0 0 0 0 1 1 1 1 1 1 1 0 0 0 0 0 1 1
0 0 0 0 0 1 0 0 0 0 0 1 1 0 0 0 0 0 0 1
0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 1 1
0 0 0 0 0 0 1 0 0 0 1 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 1 1 0 1 0 0 0 0 0 0 0 0 1 0
1 1 0 0 1 0 1 0 0 0 0 0 0 0 1 0 1 0 1 0
0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1
0 0 0 0 0 1 0 0 1 1 0 0 0 0 0 0 0 0 0 1
0 0 0 0 0 1 0 0 1 1 1 1 0 0 0 0 0 1 0 0
0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 1 1 0 1
0 1 0 1 1 0 0 1 1 0 0 0 0 1 0 0 1 0 0 1
0 1 1 0 1 0 1 1 0 1 1 1 0 0 0 0 1 1 1 0
0 0 0 0 0 0 0 0 1 0 0 1 0 0 0 1 1 1 1 0
0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 1 1 1 0
1 1 0 1 0 0 0 0 1 0 1 0 1 0 1 1 0 1 1 0
0 0 0 1 0 0 0 0 0 0 0 1 0 1 1 1 0 0 1 1
0 1 0 0 0 0 1 1 1 1 1 1 1 0 0 0 0 0 0 0
1 1 0 1 0 0 0 0 0 1 0 0 0 1 1 1 0 0 1 1
1 0 0 0 1 1 1 1 1 0 0 0 0 0 0 0 0 0 1 1
0 1 0 1 0 1 1 1 1 0 0 0 0 0 0 1 1 1 1 1

========================================
```

3. Run this program on multiple machines (2pts - Bonus)

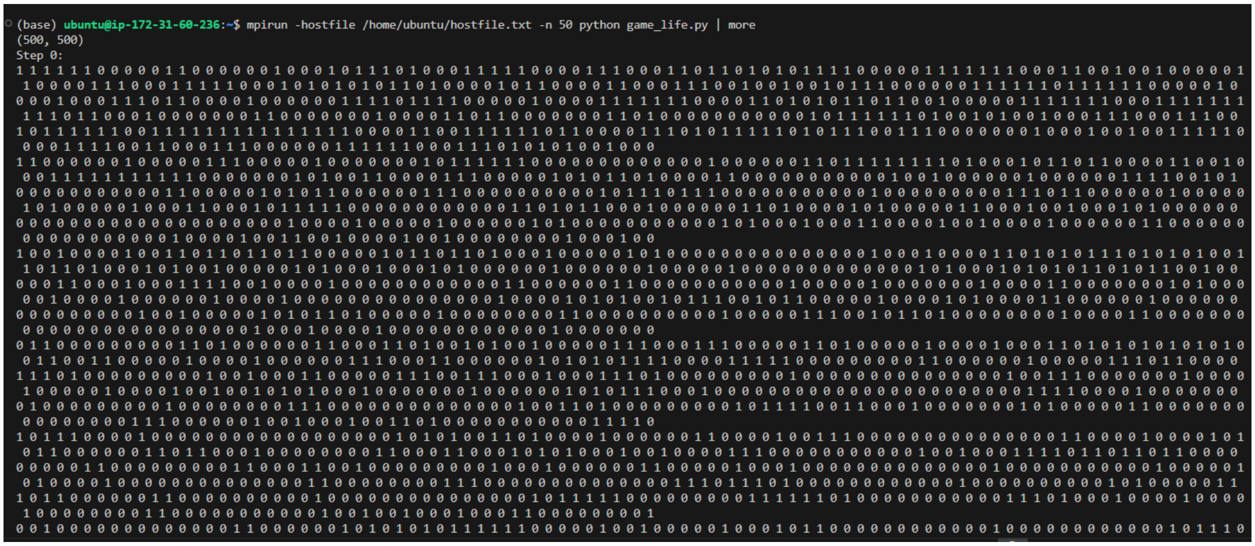
I used 6 nodes with 20 processes, each process dealing with one strip. From this experiment, one thing I observed is that the number of rows (strips) should be evenly distributed among the processes. This could be achieved by ensuring that the number of processes is divisible by the number of rows. This balances the workload;

otherwise, we can encounter errors like Segmentation Fault

```
functions_game_life.py
(base) ubuntu@ip-172-31-60-236:~$ scp functions_game_life.py ubuntu@node5:functions_game_life.py
functions_game_life.py
(base) ubuntu@ip-172-31-60-236:~$ scp functions_game_life.py ubuntu@node6:functions_game_life.py
functions_game_life.py
(base) ubuntu@ip-172-31-60-236:~$ mpirun -hostfile /home/ubuntu/hostfile.txt -n 20 python game_life.py
(20, 20)
Step 0:
0 0 0 0 0 1 1 0 1 0 1 0 1 0 0 1 1 0 0 0
0 0 1 0 0 0 1 0 0 0 0 0 0 0 0 0 0 1 0 0
0 0 0 0 1 0 1 0 1 0 0 0 0 0 0 0 1 1 0 1
0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 1 1 1 0 1
0 0 1 0 0 0 0 0 0 0 0 0 0 1 1 0 0 1 0 1
0 1 1 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 1
1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1
1 0 0 1 1 0 0 0 0 0 0 1 1 0 0 0 0 0 0 1
0 0 0 0 0 1 0 0 0 0 1 1 0 0 0 0 0 0 0 1
0 0 1 1 0 0 0 0 0 0 1 1 0 1 0 0 0 0 1 0
0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
0 1 0 1 0 0 0 0 0 0 0 1 1 0 0 0 1 0 0 0
0 1 1 0 1 0 1 0 0 0 0 0 0 0 0 0 1 1 0 0
1 0 1 1 0 0 1 0 0 0 0 1 1 0 0 0 1 0 0 0
0 0 1 1 1 0 1 1 0 0 0 0 0 0 0 0 0 0 0 1
0 0 0 0 1 1 1 1 0 0 0 0 1 0 0 0 0 0 0 1
0 0 0 0 0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 1
1 0 0 0 0 0 0 1 0 0 0 0 0 1 0 1 0 0 0 0
0 1 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 1 1 0
0 0 0 1 1 1 0 0 0 0 1 1 1 1 1 1 1 1 1 0

========================================

(20, 20)
Step 1:
0 0 0 0 0 1 1 1 0 0 0 0 0 0 0 0 1 0 0 0
0 0 0 0 0 0 1 0 0 1 0 0 0 0 0 1 0 1 1 0
0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 1 0 0 0 0
0 0 0 0 0 1 0 1 0 0 0 0 0 0 1 1 0 0 0 1
0 1 1 0 0 0 0 0 0 0 0 0 0 0 1 1 0 1 0 1
0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
1 0 1 1 0 0 0 0 0 0 0 1 1 0 0 0 0 0 1 1
0 0 0 0 1 0 0 0 0 0 1 1 1 0 0 0 0 0 0 1
0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1
0 0 1 1 1 0 0 0 0 0 1 1 1 0 0 0 0 0 1 1
0 1 0 0 1 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0
0 1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0
1 0 0 0 1 1 0 0 0 0 0 0 0 0 0 1 1 1 0 0
0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 1 1 0 0
0 1 1 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0
0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1
0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0
```

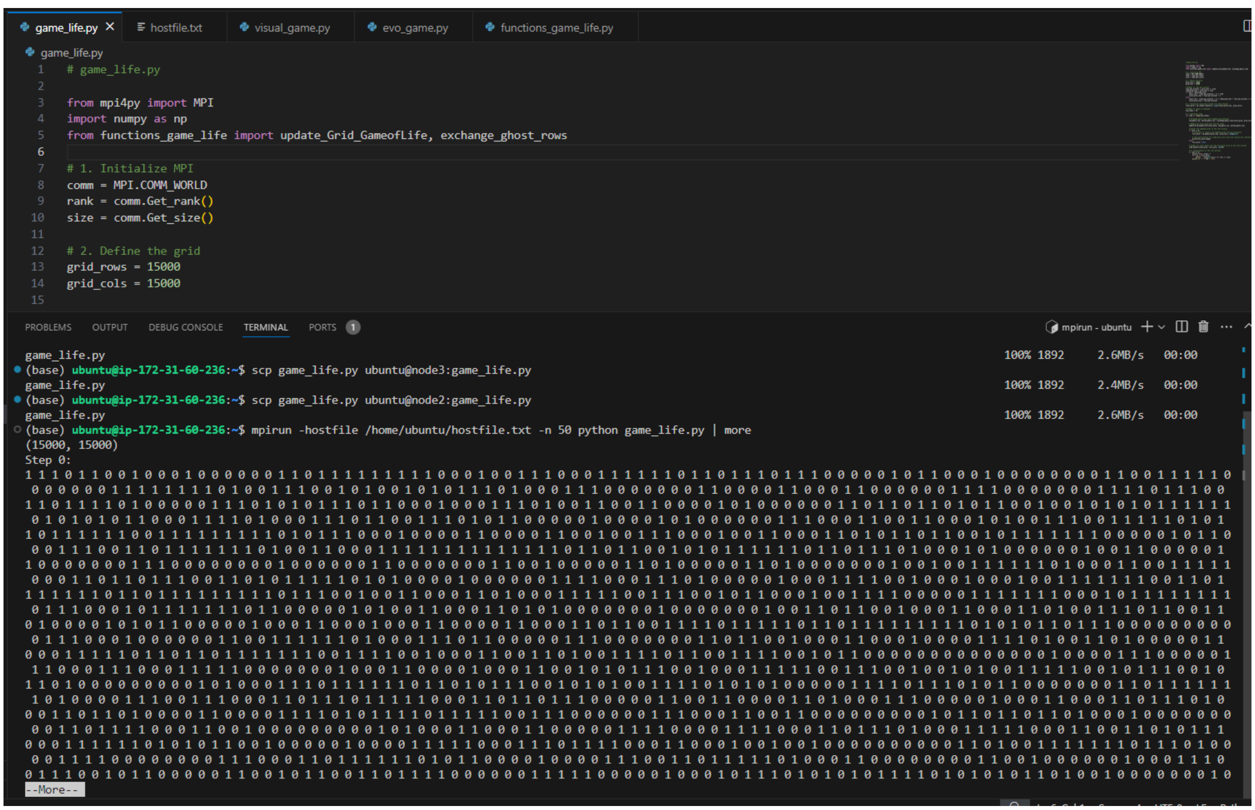
4. Run and correctly display 500x500 grid (2pts - bonus)

This is a terminal display. Since there is characters limitation, I printed the grid shape , as well I have saved the terminal in '*output_500.txt*', where you see the grid correctly.



5. What is the largest grid you can run? (5pts - bonus)

The largest grid I could run is 15000 * 15000, with 50 processes using 6 nodes including master node. At first, I tried using dimensions of 100000 by 100000 and 50000 by 50000, which wasn't successful. However, 30000 by 30000 could have been feasible, but it was taking a very long time. So, I opted for 15000 by 15000.



6. **Display the evolution of the grid step by step with screen updates (5pts - bonus)**

   I tried this using os.system for live updates, but I wasn't successful with mpi , however I did it in normal python program file *'evo_game_life.py'*

<u>Part 2: City Distances using Genetic Algorithm (60 points):</u>

1) Two function to complete in genetic_algorithms_functions.py (10 - pts):

- calculate_fitness (calculates the fitness of a given route)

```python
def calculate_fitness(route, distance_matrix):
    """
    calculate_fitness function: total distance traveled by the car.

    Parameters:
        - route (list): A list representing the order of nodes visited in the route.
        - distance_matrix (numpy.ndarray): A matrix of the distances between nodes.
            A 2D numpy array where the element at position [i, j] represents the distance between node i and node j.
    Returns:
        - float: The negative total distance traveled (negative because we want to minimize distance).
            Returns a large negative penalty if the route is infeasible.
    """
    total_distance = 0

    # Iterate through the route to calculate the total distance
    for i in range(len(route) - 1):  # Subtracted 1 to avoid index error on the last element
        node1 = route[i]
        node2 = route[i + 1]
        distance = distance_matrix[node1][node2]

        # Check if the route is infeasible
        if distance == 10000:
            return 1e6

        total_distance += distance

    total_distance += distance_matrix[route[-1]][route[0]]

    # Return the negative of total_distance because we want to minimize the distance
    return total_distance
```

- <u>Explanation of code:</u>
    - Inputs: route and distance_matrix
    - Process: initialize total distance to zero, it iterate as the length of the routes, in each iteration it takes two consecutive nodes, and calculates their distances and adds to the total distance.

    - Output: We have two returns one is if the distance is 10000,it returns 1e6 as penalty, otherwise it returns total distance.

- select_in_tournament (selecting individuals from a population for reproduction based on their fitness scores.)

```python
def select_in_tournament(population,
                         scores,
                         number_tournaments=4,
                         tournament_size=3):
    """
    Tournament selection for genetic algorithm.

    Parameters:
        - population (list): The current population of routes.
        - scores (np.array): The calculate_fitness scores corresponding to each individual in the population.
        - number_tournaments (int): The number of the tournamnents to run in the population.
        - tournament_size (int): The number of individual to compete in the tournaments.

    Returns:
        - list: A list of selected individuals for crossover.
    """
    selected = []

    for _ in range(number_tournaments):
        select_individuals = np.random.choice(range(len(population)),
                                              size=tournament_size,
                                               replace=False)

        best_idx = select_individuals[np.argmax(scores[select_individuals])]

        selected.append(population[best_idx])
    return selected
```

- <u>Explanation of code:</u>

    - Input:population, scores, number_tournaments, tournament_size
    - Process: initializes empty list to store selected individuals, loops through number of tournaments and randomly select number of individuals to compete in the tournament without repetition. Calculates the index of individual based on their fitness score.
    - Output: returns the list of selected individuals

2) Explain the program outlined in the script genetic_algorithm_trial.py and Run the Algorthim (5pts).

- <u>Explanation:</u>

    - imports libraries numpy, pandas, time and functions for genetic algorithms from genetic_algorithms_functions
    - Loads and reads csv distance matrix 'city_distances.csv' using pandas and converts it to a numpy array. The distance matrix represents the distances between cities in the TSP.

- Set parameters, population size, number of tournaments, mutation rate, number of generations, infeasible penalty, and stagnation limit.

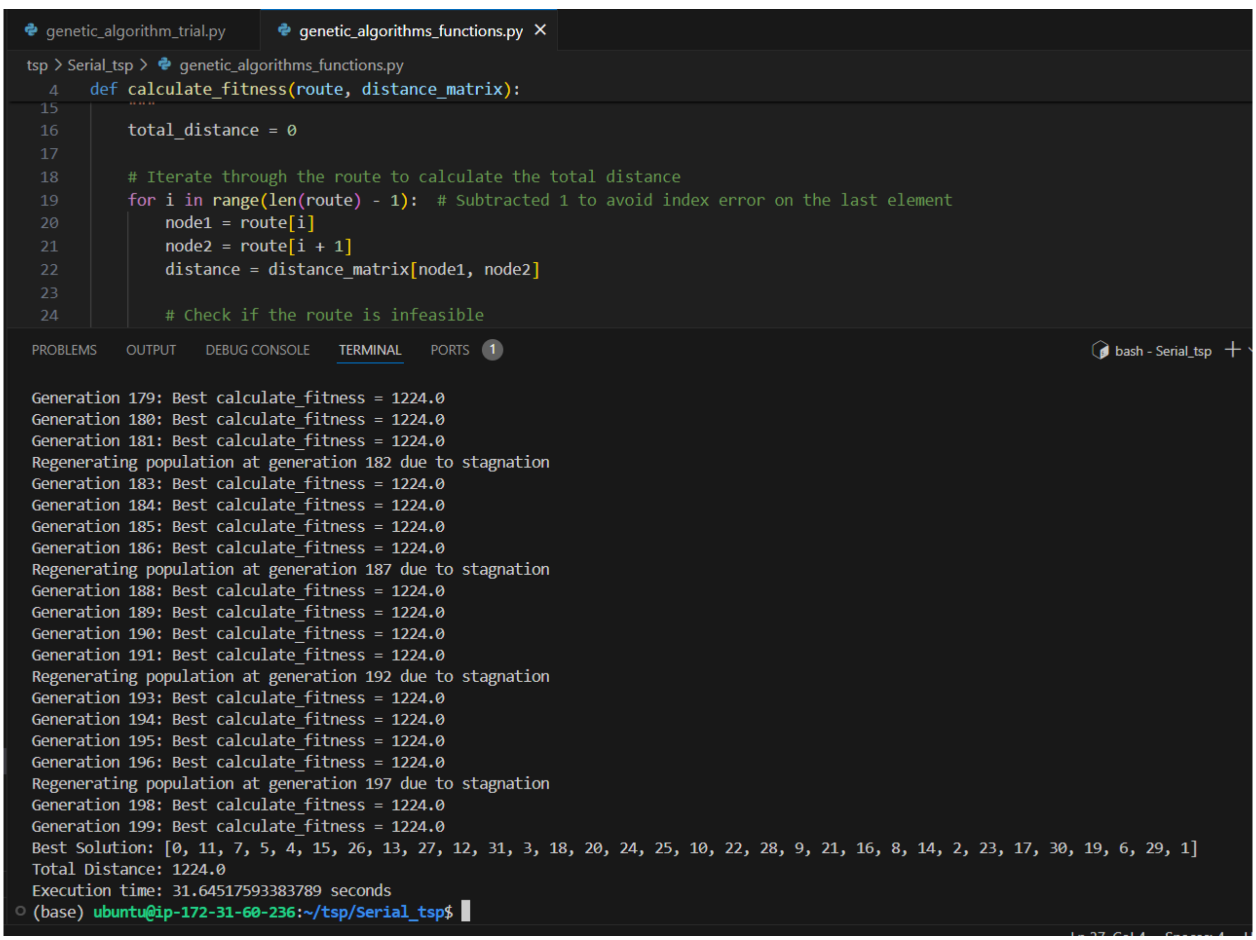
- Initial Population of routes is generated using function generate_unique_population. Each individual represents a route for the TSP

- Main GA Loop (Process):

  The program runs for a specific number of generations. In each generation, it figures out how good each route is by calculating the total distance it covers. It keeps track of the best route found so far. If there's no improvement for 5 generations, then the population gets regenerated to avoid stagnation. It then Performs selection, crossover, and mutation operations on selected individuals. Replaces individuals in the population with new offspring based on their fitness. It makes sure each route is unique, so there are no duplicates.

- Output: prints the best solution with its total distance and the execution time of the algorithm.

❖ Serial Execution time is 31.64s



3) Part6: Parallelize and distribute the code (20 pts)

Define the parts to be distributed and parallelized, explain your choices (5pts).

We need to parallelize the fitness calculation part as it is performed for each individual and is independent of the others, making it computationally intensive and most time-consuming function in the GA.

- Serial trial:

```
# Main GA loop
for generation in range(num_generations):
    # Evaluate calculate_fitness
    calculate_fitness_values = np.array([calculate_fitness(route, distance_matrix) for route in population])
```

- Parallel trial:

To parallelize efficiently I used batch processing method, along with (.delay) for asynchronous execution and (.get) to retrieve tasks results. Without the chunk method the parallelization method takes very long to complete 200 generations.

```
def chunked_routes(routes, chunk_size=20):
    for i in range(0, len(routes), chunk_size):
        yield routes[i:i + chunk_size]

for generation in range(num_generations):
    async_results = []
    for route_chunk in chunked_routes(population, 100):  # Adjust chunk size as needed

        route_chunk = [[int(node) for node in route] for route in route_chunk]
        async_results.append(calculate_fitness_async.delay(route_chunk))

    # Gather and process results
    calculate_fitness_values = np.concatenate([result.get(timeout=10) for result in async_results])

    current_best_calculate_fitness = np.min(calculate_fitness_values)
    if current_best_calculate_fitness < best_calculate_fitness:
        best_calculate_fitness = current_best_calculate_fitness
        stagnation_counter = 0
    else:
        stagnation_counter += 1
```

Use Celery to parallelize your program over one machine (10pts)

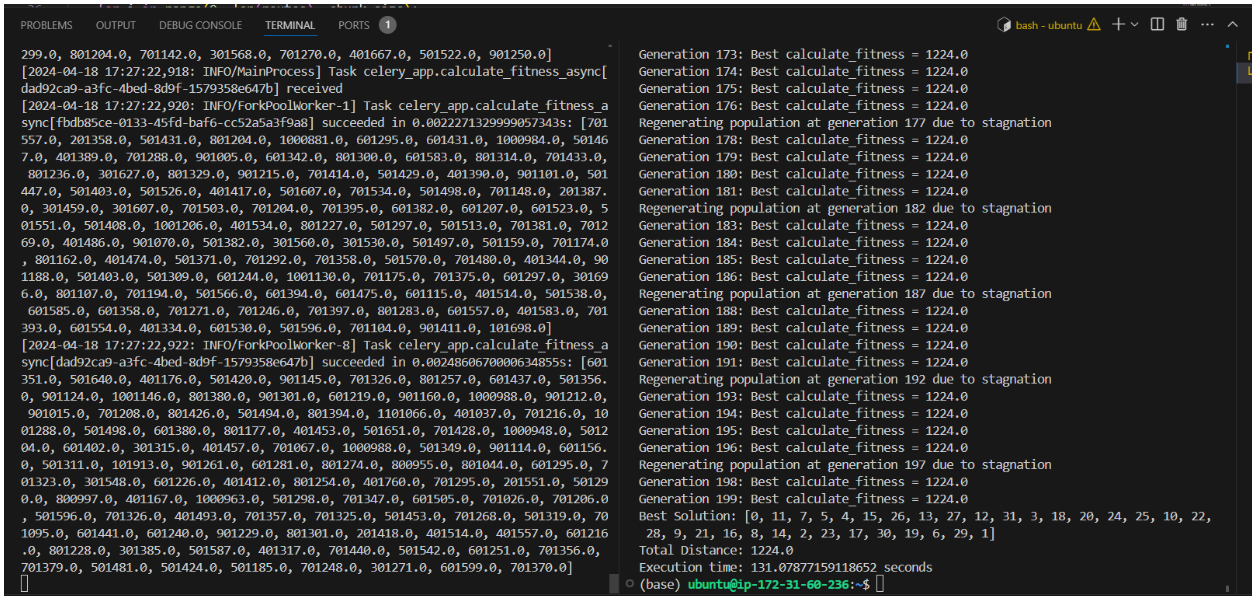
At first, I used pickle as serializer for both task and result, this approach was too slow, hence I utilized Message pack which is faster than pickle, however still slower than serial.

Single_parallel > celery_app.py

```python
# celery_app.py

from celery import Celery
from genetic_algorithms_functions import calculate_fitness
import pandas as pd


app = Celery('shortest_route',
             broker='redis://localhost:6379/0',
             backend='redis://localhost:6379/1')  # Specify the result backend

app.conf.update(
    task_serializer='msgpack',
    result_serializer='msgpack',
    accept_content=['msgpack', 'application/json'],
)

@app.task(bind=True)
def calculate_fitness_async(self, routes):
    if not hasattr(self, 'distance_matrix'):
        self.distance_matrix = pd.read_csv('city_distances.csv').values
    return [calculate_fitness(route, self.distance_matrix) for route in routes]
```

Run your code and compute the performance metrics (5pts)

- Running the code (with 10 processes):



❖ <u>Performance Metrics:</u>

- serial_time = 31.64 s
- parallel_time = 131.07 s
- num_cores = 2 (t2.large)

- speedup = serial_time / parallel_time = 31.64/131.07s = 0.241
- efficiency = speedup / num_cores = 0.241 / 2 = 0.1205

4) Part7: Enhance the algorithm (15 pts).

Distribute your algorithm over 2 machines or more using celery (7 pts).

❖ <u>To achieve it:</u>

i. I added inboud rule for redis port 6379 in security group configuration and set it to 0.0.0.0/0 to allow traffic from any IP.

ii. Configured redis to set the bind to 0.0.0.0/0 and protected mode to no

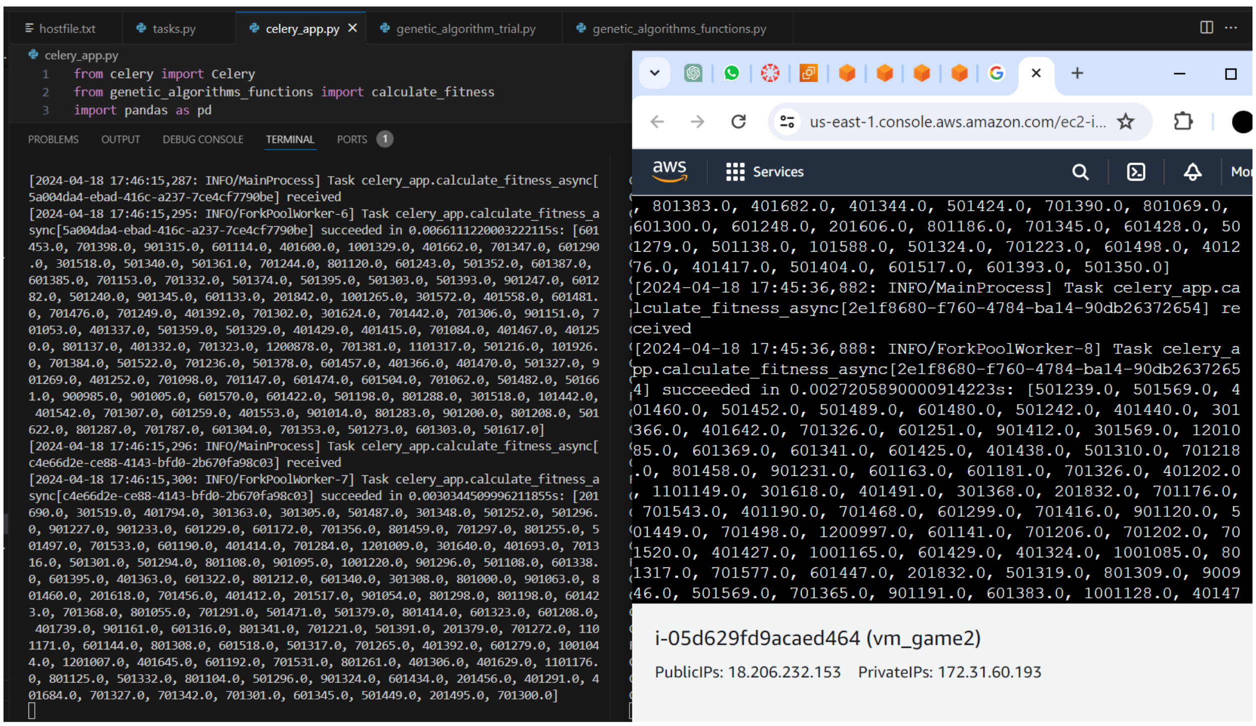
iii. changed the broker and backend from local host to private ip of the master node.

iv. SCP the files to all the 4 worker nodes

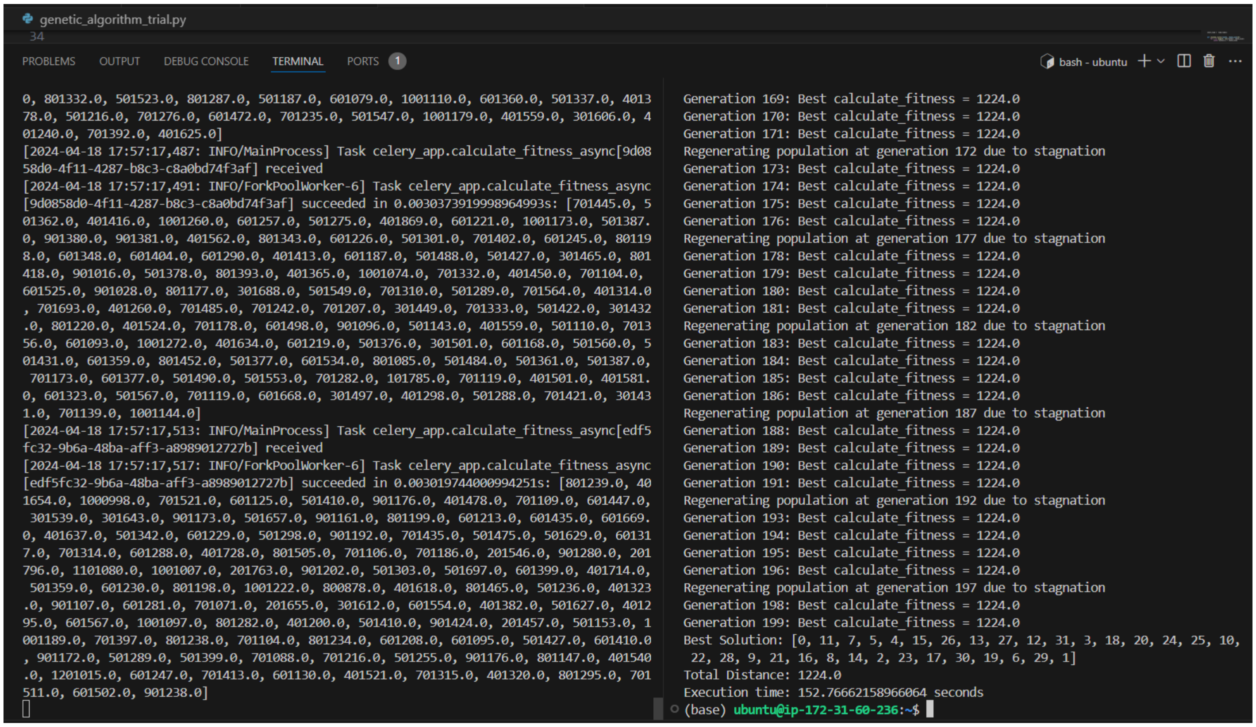
I used total of 5 nodes, each node with 10 process

```python
from celery import Celery
from genetic_algorithms_functions import calculate_fitness
import pandas as pd

app = Celery('shortest_route',
             broker='redis://172.31.60.236:6379/0',
             backend='redis://172.31.60.236:6379/0')

app.conf.update(
    task_serializer='msgpack',
    result_serializer='msgpack',
    accept_content=['msgpack', 'application/json'],
)

@app.task(bind=True)
def calculate_fitness_async(self, routes):
    if not hasattr(self, 'distance_matrix'):
        self.distance_matrix = pd.read_csv('city_distances.csv').values
    return [calculate_fitness(route, self.distance_matrix) for route in routes]
```

Here is the snip of running the program on multiple machines



Here is the output of the execution



Total parallel time take over multiple machines is 152.766s

Performance Metrics

- serial_time = 31.64 s
- parallel_time = 152.766s
- num_cores = 2 * 5 = 10 (t2.large)

- speedup = serial_time / parallel_time = 31.64/152.766s = 0.207
- efficiency = speedup / num_cores = 0.241 / 10 = 0.0207

Observation: parallel with multiple machines is slightly takes longer than parallel with one machine. Serial execution is much faster then parallel. The reason of parallel being slower could be because we are using serializer like JSON and msgpack for task inputs and outputs. Serialization can be time consuming as our distance matrix is quite large.

What improvements do you propose? (6pts)

We can make several adjustments, to enhance the algorthim by increasing the probability of reaching to global optimum.

- Instead of always randomly generating unique population, we can store the previous executions population where we found the best solution and use it as initial population, this might lead to better solution as you are already starting with best solution.

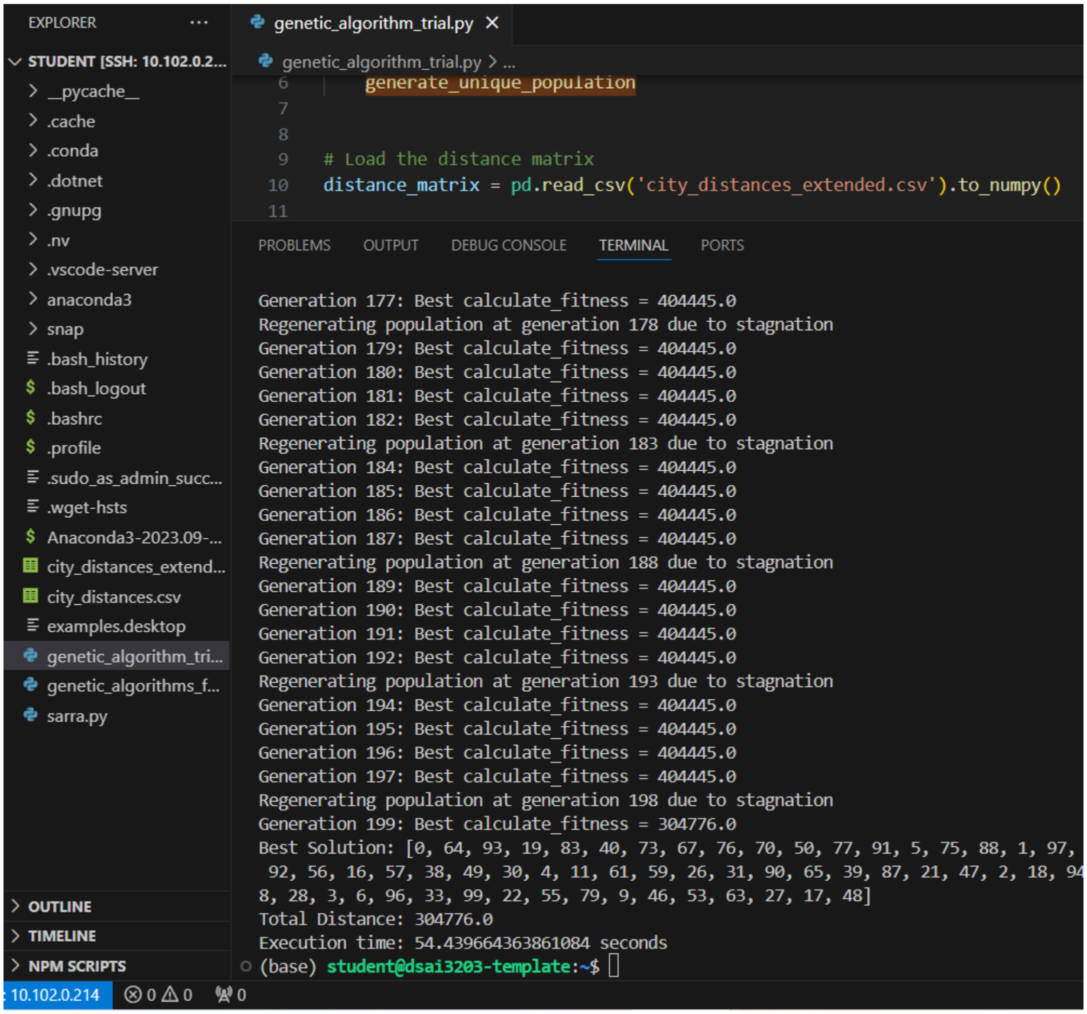
- Elitism technique: This is used to preserve the best solution and pass them to the next generation, leading to optimal solution convergence.
- Experiment with different selection methods, can help us in finding better method, and by making some adjustment or addition to that method we can optimize GA in reaching to optimal solution in shorter time.
- Using grid search to find better parameters for the algorithm.

5) Large scale problem (10 pts)

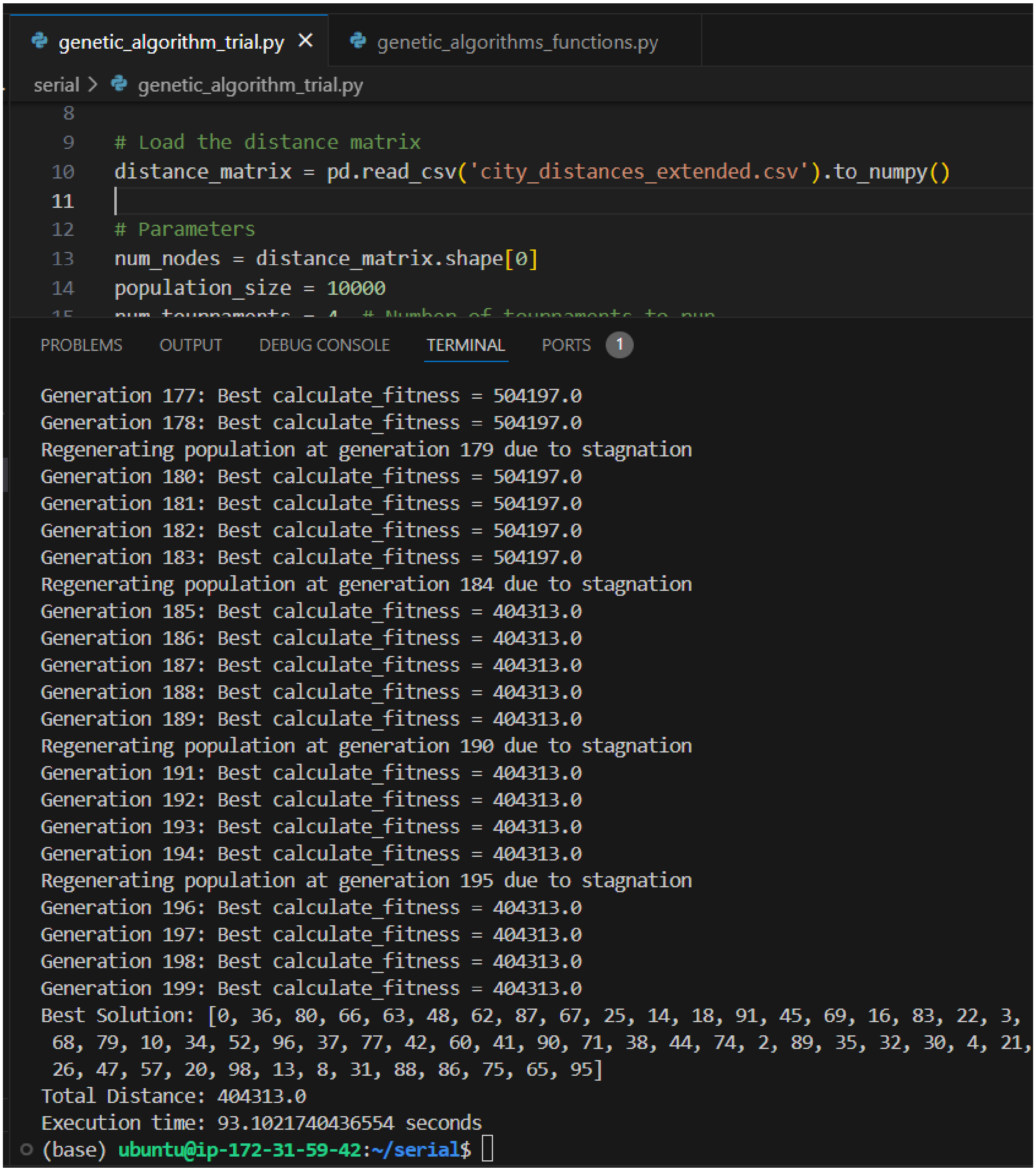
Run the program using the extended city map (5 pts):

I have noticed that Udst VM s are much faster then AWS !

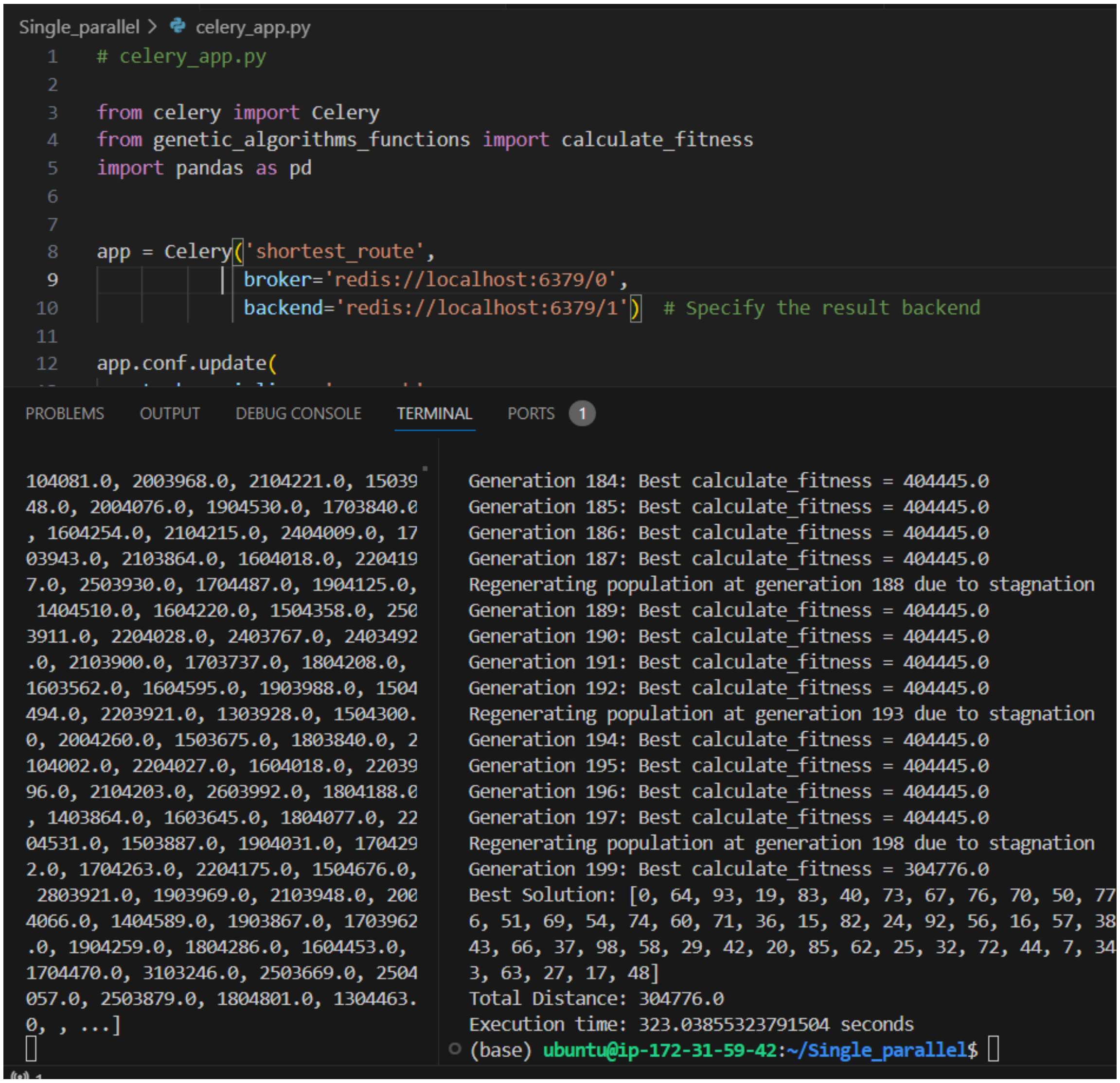
- Serial using UDST VM:



❖ Serial AWS VM:



- ❖ Parallel Aws using single machine:



```
Single_parallel > celery_app.py
# celery_app.py

from celery import Celery
from genetic_algorithms_functions import calculate_fitness
import pandas as pd


app = Celery('shortest_route',
             broker='redis://localhost:6379/0',
             backend='redis://localhost:6379/1')  # Specify the result backend

app.conf.update(
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS 1

```
104081.0, 2003968.0, 2104221.0, 15039
48.0, 2004076.0, 1904530.0, 1703840.0
, 1604254.0, 2104215.0, 2404009.0, 17
03943.0, 2103864.0, 1604018.0, 220419
7.0, 2503930.0, 1704487.0, 1904125.0,
 1404510.0, 1604220.0, 1504358.0, 250
3911.0, 2204028.0, 2403767.0, 2403492
.0, 2103900.0, 1703737.0, 1804208.0,
1603562.0, 1604595.0, 1903988.0, 1504
494.0, 2203921.0, 1303928.0, 1504300.
0, 2004260.0, 1503675.0, 1803840.0, 2
104002.0, 2204027.0, 1604018.0, 22039
96.0, 2104203.0, 2603992.0, 1804188.0
, 1403864.0, 1603645.0, 1804077.0, 22
04531.0, 1503887.0, 1904031.0, 170429
2.0, 1704263.0, 2204175.0, 1504676.0,
 2803921.0, 1903969.0, 2103948.0, 200
4066.0, 1404589.0, 1903867.0, 1703962
.0, 1904259.0, 1804286.0, 1604453.0,
1704470.0, 3103246.0, 2503669.0, 2504
057.0, 2503879.0, 1804801.0, 1304463.
0, , ...]
```

```
Generation 184: Best calculate_fitness = 404445.0
Generation 185: Best calculate_fitness = 404445.0
Generation 186: Best calculate_fitness = 404445.0
Generation 187: Best calculate_fitness = 404445.0
Regenerating population at generation 188 due to stagnation
Generation 189: Best calculate_fitness = 404445.0
Generation 190: Best calculate_fitness = 404445.0
Generation 191: Best calculate_fitness = 404445.0
Generation 192: Best calculate_fitness = 404445.0
Regenerating population at generation 193 due to stagnation
Generation 194: Best calculate_fitness = 404445.0
Generation 195: Best calculate_fitness = 404445.0
Generation 196: Best calculate_fitness = 404445.0
Generation 197: Best calculate_fitness = 404445.0
Regenerating population at generation 198 due to stagnation
Generation 199: Best calculate_fitness = 304776.0
Best Solution: [0, 64, 93, 19, 83, 40, 73, 67, 76, 70, 50, 77
6, 51, 69, 54, 74, 60, 71, 36, 15, 82, 24, 92, 56, 16, 57, 38
43, 66, 37, 98, 58, 29, 42, 20, 85, 62, 25, 32, 72, 44, 7, 34
3, 63, 27, 17, 48]
Total Distance: 304776.0
Execution time: 323.03855323791504 seconds
(base) ubuntu@ip-172-31-59-42:~/Single_parallel$
```

- ❖ Performance Metrics
  - Serial time: 93.102s
  - Parallel time: 323.03s
  - Number of cores: 2
- ▪ Speed up = 93.102s/323.03s = 0.00288
- ▪ Efficiency = 0.00288/ 2 = 0.00144

How would you add more cars to the problem? (5pts -just explain)

- ❖ <u>in our code:</u>

- we are representing each individual in the population as a single route, if we modify generate_unique_population function in a way that each individual could represent multiple routes.
- Our fitness function only calculates distance for one car. If we update the generate_unique_population function. We will also require updating the fitness to calculate the total distance traveled by all car.
- These are the two main adjustments needed according to me , but of course we will require to revise the code do other adjustments in other functions as well for more cars

Best Distance:

- City_distance (985)

genetic_algorithm_trial.py ✕ | genetic_algorithms_functions.py

serial > genetic_algorithm_trial.py

```python
infeasible_penalty = 1e6  # Penalty for infeasible routes
stagnation_limit = 5  # Number of generations without improvement bef


# Generate initial population: each individual is a route starting at
np.random.seed()  # For reproducibility
```

PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL   PORTS 1

```
Regenerating population at generation 174 due to stagnation
Generation 175: Best calculate_fitness = 985.0
Generation 176: Best calculate_fitness = 985.0
Generation 177: Best calculate_fitness = 985.0
Generation 178: Best calculate_fitness = 985.0
Regenerating population at generation 179 due to stagnation
Generation 180: Best calculate_fitness = 985.0
Generation 181: Best calculate_fitness = 985.0
Generation 182: Best calculate_fitness = 985.0
Generation 183: Best calculate_fitness = 985.0
Regenerating population at generation 184 due to stagnation
Generation 185: Best calculate_fitness = 985.0
Generation 186: Best calculate_fitness = 985.0
Generation 187: Best calculate_fitness = 985.0
Generation 188: Best calculate_fitness = 985.0
Regenerating population at generation 189 due to stagnation
Generation 190: Best calculate_fitness = 985.0
Generation 191: Best calculate_fitness = 985.0
Generation 192: Best calculate_fitness = 985.0
Generation 193: Best calculate_fitness = 985.0
Regenerating population at generation 194 due to stagnation
Generation 195: Best calculate_fitness = 985.0
Generation 196: Best calculate_fitness = 985.0
Generation 197: Best calculate_fitness = 985.0
Generation 198: Best calculate_fitness = 985.0
Regenerating population at generation 199 due to stagnation
Best Solution: [0, 8, 25, 16, 31, 27, 14, 17, 26, 2, 28, 24, 10, 19, 11, 22, 12,
Total Distance: 985.0
Execution time: 32.409419536590576 seconds
(base) ubuntu@ip-172-31-59-42:~/serial$
```

- City_distance_extended (205206)

serial > genetic_algorithm_trial.py

```python

# Parameters
num_nodes = distance_matrix.shape[0]
population_size = 10000
num_tournaments = 4  # Number of tournaments to run
mutation_rate = 0.1
num_generations = 200
infeasible_penalty = 1e6  # Penalty for infeasible routes
stagnation_limit = 5  # Number of generations without improvem


# Generate initial population: each individual is a route star
np.random.seed()  # For reproducibility
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS 1

```
Generation 184: Best calculate_fitness = 205206.0
Regenerating population at generation 185 due to stagnation
Generation 186: Best calculate_fitness = 205206.0
Generation 187: Best calculate_fitness = 205206.0
Generation 188: Best calculate_fitness = 205206.0
Generation 189: Best calculate_fitness = 205206.0
Regenerating population at generation 190 due to stagnation
Generation 191: Best calculate_fitness = 205206.0
Generation 192: Best calculate_fitness = 205206.0
Generation 193: Best calculate_fitness = 205206.0
Generation 194: Best calculate_fitness = 205206.0
Regenerating population at generation 195 due to stagnation
Generation 196: Best calculate_fitness = 205206.0
Generation 197: Best calculate_fitness = 205206.0
Generation 198: Best calculate_fitness = 205206.0
Generation 199: Best calculate_fitness = 205206.0
Best Solution: [0, 57, 69, 19, 44, 12, 41, 63, 90, 80, 93, 40, 61, 76, 5,
, 97, 28, 71, 22, 55, 77, 21, 27, 64, 59, 52, 23, 35, 37, 33, 2, 47, 31,
3, 56, 65, 68, 1, 70, 8, 60, 94, 79, 18, 86, 82, 9]
Total Distance: 205206.0
Execution time: 93.46325516700745 seconds
(base) ubuntu@ip-172-31-59-42:~/serial$
```