Nguyễn Phúc Nguyên
2274802010586

Câu 1:

Quick sort

```
import random
import time
import threading
import heapq

def generate_numbers_file(filename="numbers.txt", num_lines=25_000_000,
max_value=25_000_000):
    """Tạo file chứa 25 triệu số ngẫu nhiên."""
    with open(filename, "w") as f:
        for _ in range(num_lines):
            f.write(f"{random.randint(1, max_value)}\n")

def quicksort(arr):
    """Quicksort tuần tự."""
    if len(arr) <= 1:
        return arr
    pivot = arr[len(arr) // 2]
    left = [x for x in arr if x < pivot]
    middle = [x for x in arr if x == pivot]
    right = [x for x in arr if x > pivot]
    return quicksort(left) + middle + quicksort(right)

def parallel_quicksort(arr, num_threads=4):
    """Quicksort song song sử dụng đa luồng."""
    if len(arr) <= 1:
        return arr
    pivot = arr[len(arr) // 2]
    left = [x for x in arr if x < pivot]
    middle = [x for x in arr if x == pivot]
    right = [x for x in arr if x > pivot]

    sorted_left = []
    sorted_right = []

    left_thread = threading.Thread(target=lambda: sorted_left.extend(quicksort(left)))
    right_thread = threading.Thread(target=lambda: sorted_right.extend(quicksort(right)))

    left_thread.start()
    right_thread.start()
```

Nguyễn Phúc Nguyên
2274802010586

```
    left_thread.join()
    right_thread.join()

    return sorted_left + middle + sorted_right

def merge_sorted_chunks(sorted_chunks):
    """Hợp nhất các khối dữ liệu đã sắp xếp với dữ liệu động."""
    return list(heapq.merge(*sorted_chunks))

def read_and_sort(filename="numbers.txt", chunk_size=1_000_000):
    """Đọc file theo từng khối và sắp xếp song song."""
    sorted_chunks = []
    with open(filename, "r") as f:
        while chunk := f.readlines(chunk_size):
            numbers = list(map(int, chunk))
            sorted_chunks.append(parallel_quicksort(numbers))

    sorted_numbers = merge_sorted_chunks(sorted_chunks)
    with open("sorted_numbers.txt", "w") as f:
        for num in sorted_numbers:
            f.write(f"{num}\n")

if __name__ == "__main__":
    start_time = time.time()
    print("Đang tạo file...")
    generate_thread = threading.Thread(target=generate_numbers_file)
    generate_thread.start()
    generate_thread.join()
    print("Đã tạo xong file numbers.txt")
    end_time = time.time()
    print(f"Tổng thời gian tạo file: {end_time - start_time:.2f} giây")

    print("Bắt đầu sắp xếp...")
    sort_start_time = time.time()
    read_and_sort()
    sort_end_time = time.time()
    print(f"Tổng thời gian sắp xếp: {sort_end_time - sort_start_time:.2f} giây")
```

Nguyễn Phúc Nguyên
2274802010586

```
Đang tạo file...
Đã tạo xong file numbers.txt
Tổng thời gian tạo file: 19.92 giây
Bắt đầu sắp xếp...
Tổng thời gian sắp xếp: 62.67 giây
```

Merge Sort and Bubble Sort:

```
import random

import time
import threading
import heapq
from concurrent.futures import ThreadPoolExecutor

def generate_numbers_file(filename="numbers.txt", num_lines=25_000_000,
max_value=25_000_000):
    with open(filename, "w") as f:
        f.writelines(f"{random.randint(1, max_value)}\n" for _ in range(num_lines))

def merge_sort(arr):
    if len(arr) <= 1:
        return arr
    mid = len(arr) // 2
    left = merge_sort(arr[:mid])
    right = merge_sort(arr[mid:])
    return merge(left, right)

def quick_sort(arr):
    if len(arr) <= 1:
        return arr
    pivot = arr[len(arr) // 2]
    left = [x for x in arr if x < pivot]
    middle = [x for x in arr if x == pivot]
    right = [x for x in arr if x > pivot]
    return quick_sort(left) + middle + quick_sort(right)

def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        swapped = False
        for j in range(0, n - i - 1):
```

Nguyễn Phúc Nguyên
2274802010586

```
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
                swapped = True
        if not swapped:
            break
    return arr

def heap_sort(arr):
    heapq.heapify(arr)
    return [heapq.heappop(arr) for _ in range(len(arr))]

def merge(left, right):
    return list(heapq.merge(left, right))

def parallel_merge_sort(arr, num_threads=4):
    if len(arr) <= 1:
        return arr

    chunk_size = len(arr) // num_threads
    chunks = [arr[i * chunk_size:(i + 1) * chunk_size] for i in range(num_threads)]

    with ThreadPoolExecutor(max_workers=num_threads) as executor:
        sorted_chunks = list(executor.map(merge_sort, chunks))

    while len(sorted_chunks) > 1:
        sorted_chunks.append(merge(sorted_chunks.pop(0), sorted_chunks.pop(0)))

    return sorted_chunks[0]

def parallel_quick_sort(arr, num_threads=4):
    chunk_size = len(arr) // num_threads
    chunks = [arr[i * chunk_size:(i + 1) * chunk_size] for i in range(num_threads)]

    with ThreadPoolExecutor(max_workers=num_threads) as executor:
        sorted_chunks = list(executor.map(quick_sort, chunks))

    while len(sorted_chunks) > 1:
        sorted_chunks.append(merge(sorted_chunks.pop(0), sorted_chunks.pop(0)))

    return sorted_chunks[0]

def parallel_bubble_sort(arr, num_threads=4):
    chunk_size = len(arr) // num_threads
```

Nguyễn Phúc Nguyên
2274802010586

```
    chunks = [arr[i * chunk_size:(i + 1) * chunk_size] for i in range(num_threads)]

    with ThreadPoolExecutor(max_workers=num_threads) as executor:
        sorted_chunks = list(executor.map(bubble_sort, chunks))

    while len(sorted_chunks) > 1:
        sorted_chunks.append(merge(sorted_chunks.pop(0), sorted_chunks.pop(0)))

    return sorted_chunks[0]

def parallel_heap_sort(arr, num_threads=4):
    chunk_size = len(arr) // num_threads
    chunks = [arr[i * chunk_size:(i + 1) * chunk_size] for i in range(num_threads)]

    with ThreadPoolExecutor(max_workers=num_threads) as executor:
        sorted_chunks = list(executor.map(heap_sort, chunks))

    while len(sorted_chunks) > 1:
        sorted_chunks.append(merge(sorted_chunks.pop(0), sorted_chunks.pop(0)))

    return sorted_chunks[0]

def read_and_sort(filename="numbers.txt", chunk_size=1_000_000,
sort_method="merge"):
    sorted_chunks = []
    with open(filename, "r") as f:
        while True:
            chunk = f.readlines(chunk_size)
            if not chunk:
                break
            numbers = list(map(int, chunk))
            if sort_method == "merge":
                sorted_chunks.append(parallel_merge_sort(numbers))
            elif sort_method == "quick":
                sorted_chunks.append(parallel_quick_sort(numbers))
            elif sort_method == "bubble":
                sorted_chunks.append(parallel_bubble_sort(numbers))
            elif sort_method == "heap":
                sorted_chunks.append(parallel_heap_sort(numbers))
            else:
                raise ValueError("Phương thức sắp xếp không hợp lệ")

    sorted_numbers = list(heapq.merge(*sorted_chunks))
```

Nguyễn Phúc Nguyên
2274802010586

```
    output_filename = f"sorted_numbers_{sort_method}.txt"
    with open(output_filename, "w") as f:
        f.writelines(f"{num}\n" for num in sorted_numbers)
    print(f"Kết quả được lưu vào {output_filename}")

def main():
    start_time = time.time()
    print("Đang tạo file...")
    generate_thread = threading.Thread(target=generate_numbers_file)
    generate_thread.start()
    generate_thread.join()
    print("Đã tạo xong file numbers.txt")
    print(f"Tổng thời gian tạo file: {time.time() - start_time:.2f} giây")

    for method in ["merge", "quick", "bubble", "heap"]:
        print(f"Bắt đầu sắp xếp bằng {method}...")
        sort_start_time = time.time()
        read_and_sort(sort_method=method)
        print(f"Tổng thời gian sắp xếp ({method}): {time.time() - sort_start_time:.2f} giây")

if __name__ == "__main__":
    main()
```

```
Đang tạo file...
Đã tạo xong file numbers.txt
Tổng thời gian tạo file: 53.08 giây
Bắt đầu sắp xếp bằng merge...
Kết quả được lưu vào sorted_numbers_merge.txt
Tổng thời gian sắp xếp (merge): 361.29 giây
Bắt đầu sắp xếp bằng quick...
Kết quả được lưu vào sorted_numbers_quick.txt
Tổng thời gian sắp xếp (quick): 242.58 giây
Bắt đầu sắp xếp bằng bubble...
```

Câu 2:

```
import concurrent.futures
import threading
from collections import defaultdict
```

Nguyễn Phúc Nguyên
2274802010586

```
lock = threading.Lock()  # Để bảo vệ việc cập nhật ma trận C

# Hàm nhân ma trận CSR bằng threading
def csr_matrix_multiply(A_values, A_columns, A_row_ptr, B_values, B_columns,
B_row_ptr, A_shape, B_shape):
    C_values = []
    C_columns = []
    C_row_ptr = [0]

    with concurrent.futures.ThreadPoolExecutor() as executor:
        futures = []
        for i in range(A_shape[0]):
            futures.append(executor.submit(multiply_row, i, A_values, A_columns,
A_row_ptr, B_values, B_columns, B_row_ptr, B_shape))

        for future in concurrent.futures.as_completed(futures):
            row_values, row_columns = future.result()
            with lock:  # Bảo vệ khi cập nhật C_values và C_columns
                C_values.extend(row_values)
                C_columns.extend(row_columns)
                C_row_ptr.append(len(C_values))

    return C_values, C_columns, C_row_ptr

# Hàm nhân một hàng của A với toàn bộ ma trận B
def multiply_row(i, A_values, A_columns, A_row_ptr, B_values, B_columns,
B_row_ptr, B_shape):
    row_result = defaultdict(float)

    row_start = A_row_ptr[i]
    row_end = A_row_ptr[i+1]

    for j in range(row_start, row_end):
        A_val = A_values[j]
        A_col = A_columns[j]

        with lock:  # Đảm bảo an toàn khi đọc B_values động
            B_range_start = B_row_ptr[A_col]
            B_range_end = B_row_ptr[A_col+1]

        for k in range(B_range_start, B_range_end):
            with lock:
```

```
            B_val = B_values[k]
            B_col = B_columns[k]

        row_result[B_col] += A_val * B_val

    row_values = list(row_result.values())
    row_columns = list(row_result.keys())

    return row_values, row_columns

# Hàm kiểm tra với ma trận ví dụ
def test_matrix_multiplication():
    A_values = [1, 3, 4]
    A_columns = [0, 2, 0]
    A_row_ptr = [0, 1, 2, 3]
    A_shape = (3, 3)

    B_values = [1, 2, 5]
    B_columns = [0, 1, 2]
    B_row_ptr = [0, 1, 2, 3]
    B_shape = (3, 3)

    C_values, C_columns, C_row_ptr = csr_matrix_multiply(A_values, A_columns,
A_row_ptr, B_values, B_columns, B_row_ptr, A_shape, B_shape)

    print("C_values:", C_values)
    print("C_columns:", C_columns)
    print("C_row_ptr:", C_row_ptr)

test_matrix_multiplication()
```

```
C_values: [15.0, 1.0, 4.0]
C_columns: [2, 0, 0]
C_row_ptr: [0, 1, 2, 3]
```

Câu 3:

```
import numpy as np
from PIL import Image
from concurrent.futures import ThreadPoolExecutor
```

Nguyễn Phúc Nguyên
2274802010586

```
# Chia ảnh thành các khối nhỏ đồng nhất
def split_image(image, block_size=(100, 100)):
    image_array = np.array(image)
    blocks = []
    h, w = image_array.shape[:2]

    # Chia ảnh thành các khối 100x100 (hoặc kích thước khác)
    for i in range(0, h, block_size[0]):
        for j in range(0, w, block_size[1]):
            block = image_array[i:i+block_size[0], j:j+block_size[1]]

            # Thêm padding nếu khối nhỏ hơn kích thước yêu cầu
            if block.shape[0] < block_size[0] or block.shape[1] < block_size[1]:
                padded_block = np.zeros((block_size[0], block_size[1], 3), dtype=block.dtype)
                padded_block[:block.shape[0], :block.shape[1]] = block
                blocks.append(padded_block)
            else:
                blocks.append(block)

    return blocks

# Áp dụng bộ lọc cho từng khối ảnh
def apply_filter_to_block(block, filter_params):
    # Ví dụ áp dụng bộ lọc thay đổi độ sáng
    brightness = filter_params['brightness']
    return block * brightness

# Xử lý ảnh song song
def process_image_parallel(image, filter_params):
    blocks = split_image(image)

    # Sử dụng ThreadPoolExecutor để xử lý các khối ảnh song song
    with ThreadPoolExecutor() as executor:
        processed_blocks = list(executor.map(lambda block: apply_filter_to_block(block,
filter_params), blocks))

    # Kết hợp các khối lại với nhau
    block_size = processed_blocks[0].shape[0]  # Lấy kích thước của khối (đảm bảo tất cả
các khối có cùng kích thước)

    # Tính số lượng khối trên mỗi hàng (số cột)
    num_blocks_per_row = image.width // block_size
```

Nguyễn Phúc Nguyên
2274802010586

```
    rows = []
    for i in range(0, len(processed_blocks), num_blocks_per_row):
        row_blocks = processed_blocks[i:i + num_blocks_per_row]

        # Kiểm tra nếu số lượng khối trong hàng cuối không đủ, thêm padding cho đủ
        if len(row_blocks) < num_blocks_per_row:
            row_blocks += [np.zeros_like(row_blocks[0])] * (num_blocks_per_row -
len(row_blocks))

        row = np.concatenate(row_blocks, axis=1)
        rows.append(row)

    # Kết hợp các hàng lại với nhau
    return np.concatenate(rows, axis=0)

# Đọc ảnh và áp dụng bộ lọc
image = Image.open('image.jpg')
filter_params = {'brightness': 1.2}  # Ví dụ về bộ lọc
result_image = process_image_parallel(image, filter_params)

# Chuyển đổi lại thành ảnh để hiển thị
result_image_pil = Image.fromarray(result_image.astype(np.uint8))
result_image_pil.show()
```

Câu 4:

```
# Câu 4:
import heapq
import random
from concurrent.futures import ThreadPoolExecutor, as_completed
import threading

# Hàm Dijkstra
def dijkstra_parallel(graph, start_vertex):
    # Số đỉnh trong đồ thị
    num_vertices = len(graph)

    # Khoảng cách ban đầu từ start_vertex
    distances = {vertex: float('inf') for vertex in range(num_vertices)}
```

Nguyễn Phúc Nguyên
2274802010586

```
    distances[start_vertex] = 0

    # Priority Queue (Min-heap)
    min_heap = [(0, start_vertex)]  # (distance, vertex)
    heapq.heapify(min_heap)

    # Đảm bảo tính đồng bộ khi truy cập và thay đổi các phần tử
    lock = threading.Lock()

    # Hàm xử lý mỗi đỉnh trong đồ thị
    def process_vertex(vertex):
        with lock:
            # Lấy đỉnh có khoảng cách tối thiểu
            distance, current_vertex = heapq.heappop(min_heap)
            if distance > distances[current_vertex]:
                return

            # Cập nhật các đỉnh kề
            for neighbor, weight in graph[current_vertex]:
                new_distance = distances[current_vertex] + weight
                if new_distance < distances[neighbor]:
                    distances[neighbor] = new_distance
                    heapq.heappush(min_heap, (new_distance, neighbor))

    # Sử dụng ThreadPoolExecutor để xử lý song song
    with ThreadPoolExecutor() as executor:
        futures = []
        for vertex in range(num_vertices):
            futures.append(executor.submit(process_vertex, vertex))

        # Đảm bảo tất cả các thread hoàn thành
        for future in as_completed(futures):
            future.result()

    return distances

# Hàm thêm cạnh vào đồ thị
def add_edge(graph, u, v, weight):
    graph[u].append((v, weight))
    graph[v].append((u, weight))

# Hàm xóa cạnh khỏi đồ thị
def remove_edge(graph, u, v):
```

Nguyễn Phúc Nguyên
2274802010586

```
    graph[u] = [edge for edge in graph[u] if edge[0] != v]
    graph[v] = [edge for edge in graph[v] if edge[0] != u]

# Tạo đồ thị mẫu
def create_graph(num_vertices):
    graph = {i: [] for i in range(num_vertices)}
    for _ in range(100):
        u, v = random.sample(range(num_vertices), 2)
        weight = random.randint(1, 10)
        add_edge(graph, u, v, weight)
    return graph

# Sử dụng thuật toán Dijkstra song song
if __name__ == "__main__":
    num_vertices = 10  # Số đỉnh trong đồ thị
    graph = create_graph(num_vertices)

    print("Khoảng cách ban đầu:")
    print(dijkstra_parallel(graph, 0))

    # Thêm và xóa các cạnh
    print("\nSau khi thêm cạnh:")
    add_edge(graph, 0, 4, 5)
    print(dijkstra_parallel(graph, 0))

    print("\nSau khi xóa cạnh:")
    remove_edge(graph, 0, 4)
    print(dijkstra_parallel(graph, 0))
```

```
Khoảng cách ban đầu:
{0: 0, 1: 3, 2: 4, 3: 2, 4: 1, 5: 6, 6: 4, 7: 5, 8: 3, 9: 4}

Sau khi thêm cạnh:
{0: 0, 1: 3, 2: 4, 3: 2, 4: 1, 5: 6, 6: 4, 7: 5, 8: 3, 9: 4}

Sau khi xóa cạnh:
{0: 0, 1: 5, 2: 4, 3: 2, 4: 4, 5: 7, 6: 4, 7: 5, 8: 3, 9: 6}
```

Câu 5:

Nguyễn Phúc Nguyên
2274802010586

```python
#CÂU 5:
import random
import math
from concurrent.futures import ThreadPoolExecutor, as_completed

# Hàm tính số điểm trong hình tròn
def monte_carlo_simulation(num_samples):
    inside_circle = 0

    # Thực hiện mô phỏng cho từng điểm ngẫu nhiên
    for _ in range(num_samples):
        x = random.uniform(-1, 1)
        y = random.uniform(-1, 1)

        # Kiểm tra xem điểm có nằm trong hình tròn không
        if x**2 + y**2 <= 1:
            inside_circle += 1

    return inside_circle

# Hàm chính để tính π song song với phản hồi thời gian thực
def estimate_pi_parallel(total_samples, feedback_interval=1000, max_threads=4):
    samples_per_thread = total_samples // max_threads
    results = []

    # Sử dụng ThreadPoolExecutor để thực hiện song song
    with ThreadPoolExecutor(max_workers=max_threads) as executor:
        futures = []

        # Chia công việc cho các thread
        for i in range(max_threads):
            futures.append(executor.submit(monte_carlo_simulation, samples_per_thread))

        inside_circle_total = 0
        for idx, future in enumerate(as_completed(futures)):
            inside_circle_total += future.result()

            # Cập nhật giá trị π và điều chỉnh chiến lược lấy mẫu sau mỗi feedback_interval
            if (idx + 1) * samples_per_thread >= feedback_interval:
                pi_estimate = 4 * inside_circle_total / ((idx + 1) * samples_per_thread)
                print(f"Feedback: ước tính giá trị của π sau {idx + 1} phần công việc là:
{pi_estimate}")
                # Cải thiện tốc độ hội tụ (có thể thay đổi chiến lược lấy mẫu ở đây nếu cần)
```

Nguyễn Phúc Nguyên
2274802010586

```
    pi_final = 4 * inside_circle_total / total_samples
    print(f"Giá trị cuối cùng của π: {pi_final}")
    return pi_final

# Ví dụ sử dụng
total_samples = 10000000  # Tổng số mẫu muốn sử dụng
estimate_pi_parallel(total_samples)
```

```
Feedback: ước tính giá trị của π sau 1 phần công việc là: 3.1410432
Feedback: ước tính giá trị của π sau 2 phần công việc là: 3.1400448
Feedback: ước tính giá trị của π sau 3 phần công việc là: 3.1411328
Feedback: ước tính giá trị của π sau 4 phần công việc là: 3.1415144
Giá trị cuối cùng của π: 3.1415144

3.1415144
```

Câu 6:

```
import zlib
import threading
from concurrent.futures import ThreadPoolExecutor
import random
import os

# Hàm nén dữ liệu sử dụng RLE (Run-Length Encoding)
def run_length_encode(data):
    encoded = []
    i = 0
    while i < len(data):
        count = 1
        while i + 1 < len(data) and data[i] == data[i + 1]:
            i += 1
            count += 1
        encoded.append((data[i], count))
        i += 1
    return encoded

# Hàm kiểm tra mức độ dư thừa trong dữ liệu
def is_data_redundant(data):
    freq = {}
    for byte in data:
```

Nguyễn Phúc Nguyên
2274802010586

```
        if byte in freq:
            freq[byte] += 1
        else:
            freq[byte] = 1

    # Nếu một ký tự chiếm hơn 50% dữ liệu, ta cho rằng có dư thừa
    max_freq = max(freq.values())
    return max_freq > len(data) / 2

# Hàm xử lý một khối dữ liệu và nén nó
def process_chunk(data):
    if is_data_redundant(data):
        # Nén dữ liệu bằng RLE nếu dữ liệu có dư thừa
        return run_length_encode(data)
    else:
        # Nếu dữ liệu không dư thừa, sử dụng nén gzip (DEFLATE)
        return zlib.compress(data)

# Hàm nén tệp song song
def parallel_file_compression(input_file, output_file, chunk_size=1024, max_threads=4):
    with open(input_file, 'rb') as infile, open(output_file, 'wb') as outfile:
        data = infile.read(chunk_size)

        with ThreadPoolExecutor(max_workers=max_threads) as executor:
            futures = []
            while data:
                futures.append(executor.submit(process_chunk, data))
                data = infile.read(chunk_size)

            for future in futures:
                compressed_chunk = future.result()
                if isinstance(compressed_chunk, bytes):
                    outfile.write(compressed_chunk)
                else:
                    # Nếu kết quả là dạng dữ liệu (ví dụ: RLE) ta cần phải chuyển thành byte
stream
                    outfile.write(bytes(str(compressed_chunk), 'utf-8'))

# Ví dụ sử dụng
if __name__ == "__main__":
    input_file = 'input.txt'  # Đảm bảo rằng tệp này tồn tại
    output_file = 'compressed_output.bin'
```

```
    parallel_file_compression(input_file, output_file)
    print(f"Tệp đã được nén và lưu vào {output_file}")
```

Câu 7:

```
#Cau 7
import random
import time

# Hàm tính tổng khoảng cách của một hành trình
def calculate_distance(tour, distance_matrix):
    total_distance = 0
    for i in range(len(tour) - 1):
        total_distance += distance_matrix[tour[i]][tour[i + 1]]
    total_distance += distance_matrix[tour[-1]][tour[0]]  # Quay lại điểm xuất phát
    return total_distance

# Hàm tạo quần thể ban đầu
def create_initial_population(num_cities, population_size):
    population = []
    for _ in range(population_size):
        tour = list(range(num_cities))
        random.shuffle(tour)
        population.append(tour)
    return population

# Hàm lai ghép (crossover)
def crossover(parent1, parent2):
    child = [-1] * len(parent1)
    start, end = sorted(random.sample(range(len(parent1)), 2))
    child[start:end] = parent1[start:end]
    for i in range(len(parent2)):
        if parent2[i] not in child:
            for j in range(len(child)):
                if child[j] == -1:
                    child[j] = parent2[i]
                    break
    return child

# Hàm đột biến (mutation)
```

Nguyễn Phúc Nguyên
2274802010586

```
def mutate(tour, mutation_rate):
    if random.random() < mutation_rate:
        idx1, idx2 = random.sample(range(len(tour)), 2)
        tour[idx1], tour[idx2] = tour[idx2], tour[idx1]
    return tour

# Thuật toán di truyền song song đơn giản
def parallel_genetic_algorithm(distance_matrix, population_size=100, generations=100,
mutation_rate=0.01):
    num_cities = len(distance_matrix)
    population = create_initial_population(num_cities, population_size)

    for generation in range(generations):
        # Đánh giá độ thích nghi của từng cá thể
        fitness_scores = [calculate_distance(tour, distance_matrix) for tour in population]

        # Chọn lọc cá thể tốt nhất
        best_tour = population[fitness_scores.index(min(fitness_scores))]

        # Tạo quần thể mới
        new_population = [best_tour]  # Giữ lại cá thể tốt nhất
        while len(new_population) < population_size:
            parent1, parent2 = random.choices(population, k=2)
            child = crossover(parent1, parent2)
            child = mutate(child, mutation_rate)
            new_population.append(child)

        population = new_population

    return best_tour, calculate_distance(best_tour, distance_matrix)

# Ví dụ sử dụng
if __name__ == "__main__":
    # Ma trận khoảng cách giữa các thành phố
    distance_matrix = [
        [0, 10, 15, 20],
        [10, 0, 35, 25],
        [15, 35, 0, 30],
        [20, 25, 30, 0]
    ]

    start_time = time.time()
    best_tour, best_distance = parallel_genetic_algorithm(distance_matrix)
```

```
    end_time = time.time()

    print(f"Best Tour: {best_tour}")
    print(f"Best Distance: {best_distance}")
    print(f"Time: {end_time - start_time} seconds")
```

```
Best Tour: [3, 1, 0, 2]
Best Distance: 80
Time: 0.0338900089263916 seconds
```

Câu 8:

```
import numpy as np
import concurrent.futures
import time
import random

# Thông số mô phỏng
GRID_SIZE = 100  # Lưới 100x100
TIME_STEPS = 50  # Số bước thời gian
VISCOSITY = 0.1  # Độ nhớt của chất lỏng

# Khởi tạo lưới vận tốc (u, v) và áp suất p
velocity_u = np.zeros((GRID_SIZE, GRID_SIZE))
velocity_v = np.zeros((GRID_SIZE, GRID_SIZE))
pressure = np.zeros((GRID_SIZE, GRID_SIZE))

# Danh sách vật cản động (sẽ thay đổi theo thời gian)
obstacles = set()

# Hàm cập nhật vận tốc theo phương trình Navier-Stokes (đơn giản hóa)
def update_velocity(x, y):
    if (x, y) in obstacles:
        return 0, 0  # Nếu là vật cản, vận tốc = 0

    u_new = velocity_u[x, y] + VISCOSITY * (velocity_u[(x-1) % GRID_SIZE, y] +
velocity_u[(x+1) % GRID_SIZE, y] +
                                            velocity_u[x, (y-1) % GRID_SIZE] + velocity_u[x, (y+1) %
GRID_SIZE] -
                                            4 * velocity_u[x, y])
```

Nguyễn Phúc Nguyên
2274802010586

```
    v_new = velocity_v[x, y] + VISCOSITY * (velocity_v[(x-1) % GRID_SIZE, y] +
velocity_v[(x+1) % GRID_SIZE, y] +
                                velocity_v[x, (y-1) % GRID_SIZE] + velocity_v[x, (y+1) %
GRID_SIZE] -
                                4 * velocity_v[x, y])

    return u_new, v_new

# Hàm cập nhật áp suất
def update_pressure(x, y):
    if (x, y) in obstacles:
        return pressure[x, y]

    return (pressure[(x-1) % GRID_SIZE, y] + pressure[(x+1) % GRID_SIZE, y] +
            pressure[x, (y-1) % GRID_SIZE] + pressure[x, (y+1) % GRID_SIZE]) / 4

# Cập nhật vật cản động (có thể xuất hiện/ngẫu nhiên thay đổi)
def update_obstacles():
    if random.random() < 0.1:  # Xác suất thay đổi vật cản 10%
        new_x = random.randint(10, GRID_SIZE - 10)
        new_y = random.randint(10, GRID_SIZE - 10)
        if (new_x, new_y) in obstacles:
            obstacles.remove((new_x, new_y))  # Loại bỏ vật cản cũ
        else:
            obstacles.add((new_x, new_y))  # Thêm vật cản mới

# Hàm chạy mô phỏng song song
def run_simulation():
    global velocity_u, velocity_v, pressure

    for step in range(TIME_STEPS):
        update_obstacles()  # Cập nhật vật cản động

        # Cập nhật vận tốc song song
        with concurrent.futures.ThreadPoolExecutor() as executor:
            future_velocities = {(x, y): executor.submit(update_velocity, x, y)
                                 for x in range(GRID_SIZE) for y in range(GRID_SIZE)}

        for (x, y), future in future_velocities.items():
            velocity_u[x, y], velocity_v[x, y] = future.result()

        # Cập nhật áp suất song song
        with concurrent.futures.ThreadPoolExecutor() as executor:
```

Nguyễn Phúc Nguyên
2274802010586

```
            future_pressures = {(x, y): executor.submit(update_pressure, x, y)
        for x in range(GRID_SIZE) for y in range(GRID_SIZE)}

        for (x, y), future in future_pressures.items():
            pressure[x, y] = future.result()

        print(f"Step {step + 1}/{TIME_STEPS} completed.")

# Chạy mô phỏng
start_time = time.time()
run_simulation()
end_time = time.time()

print(f"Mô phỏng hoàn tất sau {end_time - start_time:.2f} giây.")
```

Nguyễn Phúc Nguyên
2274802010586

```
Output exceeds the size limit. Open the full output data in a text editor
Step 1/50 completed.
Step 2/50 completed.
Step 3/50 completed.
Step 4/50 completed.
Step 5/50 completed.
Step 6/50 completed.
Step 7/50 completed.
Step 8/50 completed.
Step 9/50 completed.
Step 10/50 completed.
Step 11/50 completed.
Step 12/50 completed.
Step 13/50 completed.
Step 14/50 completed.
Step 15/50 completed.
Step 16/50 completed.
Step 17/50 completed.
Step 18/50 completed.
Step 19/50 completed.
Step 20/50 completed.
Step 21/50 completed.
Step 22/50 completed.
Step 23/50 completed.
Step 24/50 completed.
Step 25/50 completed.
...
Step 48/50 completed.
Step 49/50 completed.
Step 50/50 completed.
Mô phỏng hoàn tất sau 19.75 giây.
```

Cau 9:

```python
#Cau 9
import random
import time

# Hàm tìm các tập phổ biến
```

Nguyễn Phúc Nguyên
2274802010586

```
def find_frequent_itemsets(transactions, min_support):
    item_counts = {}
    for transaction in transactions:
        for item in transaction:
            item_counts[item] = item_counts.get(item, 0) + 1

    frequent_itemsets = {frozenset([item]): count for item, count in item_counts.items() if
count >= min_support}
    return frequent_itemsets

# Hàm sinh các tập ứng viên mới
def generate_candidates(prev_itemsets, k):
    candidates = set()
    for itemset1 in prev_itemsets:
        for itemset2 in prev_itemsets:
            union = itemset1.union(itemset2)
            if len(union) == k:
                candidates.add(union)
    return candidates

# Thuật toán Apriori song song đơn giản
def parallel_apriori(transactions, min_support=2):
    frequent_itemsets = find_frequent_itemsets(transactions, min_support)
    k = 2

    while True:
        candidates = generate_candidates(frequent_itemsets.keys(), k)
        if not candidates:
            break

        candidate_counts = {candidate: 0 for candidate in candidates}
        for transaction in transactions:
            for candidate in candidates:
                if candidate.issubset(transaction):
                    candidate_counts[candidate] += 1

        new_frequent_itemsets = {itemset: count for itemset, count in
candidate_counts.items() if count >= min_support}
        if not new_frequent_itemsets:
            break

        frequent_itemsets.update(new_frequent_itemsets)
        k += 1
```

```
    return frequent_itemsets

# Ví dụ sử dụng
if __name__ == "__main__":
    transactions = [
        {'A', 'B', 'C'},
        {'A', 'B'},
        {'A', 'C'},
        {'A'},
        {'B', 'C'},
        {'B'},
        {'C'}
    ]

    start_time = time.time()
    frequent_itemsets = parallel_apriori(transactions, min_support=2)
    end_time = time.time()

    print("Frequent Itemsets:")
    for itemset, count in frequent_itemsets.items():
        print(f"{itemset}: {count}")
    print(f"Time: {end_time - start_time} seconds")
```

```
Frequent Itemsets:
frozenset({'A'}): 4
frozenset({'C'}): 4
frozenset({'B'}): 4
frozenset({'A', 'B'}): 2
frozenset({'A', 'C'}): 2
frozenset({'C', 'B'}): 2
Time: 0.0 seconds
```

Cau 10:

```
#Cau 10
import random
import time

# Hàm tính đầu ra của mạng neural
def neural_network(input_data, weights):
```

Nguyễn Phúc Nguyên
2274802010586

```
    return sum(x * w for x, w in zip(input_data, weights))

# Hàm cập nhật trọng số
def update_weights(weights, input_data, target, learning_rate=0.01):
    prediction = neural_network(input_data, weights)
    error = target - prediction
    for i in range(len(weights)):
        weights[i] += learning_rate * error * input_data[i]
    return weights

# Thuật toán huấn luyện mạng neural song song đơn giản
def parallel_neural_network(training_data, weights, epochs=100, learning_rate=0.01):
    for epoch in range(epochs):
        for input_data, target in training_data:
            weights = update_weights(weights, input_data, target, learning_rate)
    return weights

# Ví dụ sử dụng
if __name__ == "__main__":
    training_data = [
        ([0, 0], 0),
        ([0, 1], 1),
        ([1, 0], 1),
        ([1, 1], 0)
    ]
    weights = [random.random() for _ in range(2)]

    start_time = time.time()
    trained_weights = parallel_neural_network(training_data, weights)
    end_time = time.time()

    print(f"Trained Weights: {trained_weights}")
    print(f"Time: {end_time - start_time} seconds")
```

```
Trained Weights: [0.4481127040612886, 0.22694934842709225]
Time: 0.0 seconds
```