<!--Parallel and Concurrent Programming-->

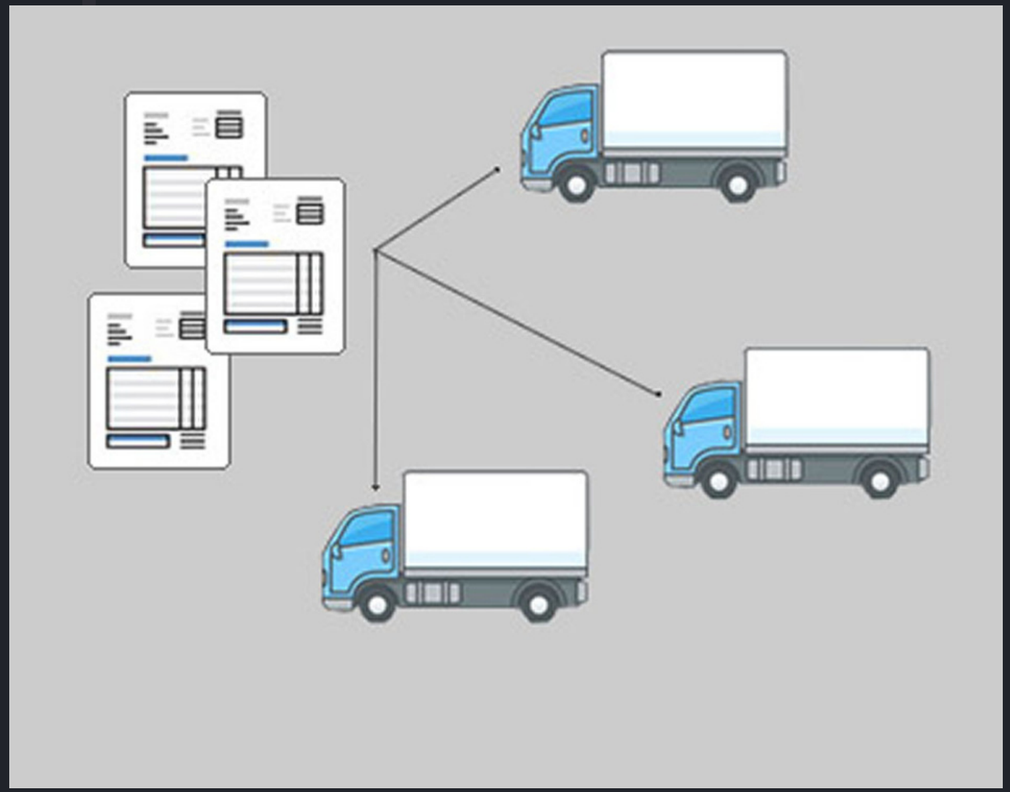
Optimizing Delivery Routes for Distribution Companies Using Parallel Merge Sort

}

Final Project
<By="Jatziry Sanchez Wong and Fiorella Mirano Surquislla"/>

Introduction {

In today's world of e-commerce, efficient delivery logistics is critical to customer satisfaction and the success of distribution companies. This project addresses the challenge of optimizing delivery routes for companies like Amazon, AliExpress and Shein, which handle large volumes of orders. To achieve this, we implement a parallel sorting algorithm, specifically Merge Sort, which sorts delivery destinations based on distance from the point of origin, thus maximizing delivery efficiency.



}

Let's explain the code!{

Let's explain the code!{

```python
import sqlite3 #gestión de la base de datos
import random #generación de dummy data
import threading #hilos
import time #medir tiempo
import os #manipulación de archivos

class Location:
    def __init__(self, id, distance_from_origin):
        self.id = id
        self.distance_from_origin = distance_from_origin

    def __lt__(self, other):
        return self.distance_from_origin < other.distance_from_origin

def create_table(cursor):
    cursor.execute('''CREATE TABLE IF NOT EXISTS clientes (
                      id INTEGER PRIMARY KEY,
                      distance_from_origin INTEGER)''')

def delete_table(cursor):
    cursor.execute('''DROP TABLE clientes''')

def insert_random_locations(cursor, num_locations):
    for i in range(num_locations):
        distance_from_origin = random.randint(0, 100)
        cursor.execute("INSERT INTO clientes (id, distance_from_origin) VALUES (?, ?)", (i, distance_from_origin))

def load_locations_from_database(cursor):
    cursor.execute("SELECT * FROM clientes")
    locations = []
    for row in cursor.fetchall():
        id, distance_from_origin = row
        locations.append(Location(id, distance_from_origin))
    return locations
```

Let's explain the code!{

```python
# number of elements
MAX = 2000000

# number of threads
THREAD_MAX = 8

def merge(arr, low, mid, high):
    left = arr[low:mid+1]
    right = arr[mid+1:high+1]

    n1 = len(left)
    n2 = len(right)
    i = j = 0
    k = low

    while i < n1 and j < n2:
        if left[i] < right[j]:
            arr[k] = left[i]
            i += 1
        else:
            arr[k] = right[j]
            j += 1
        k += 1

    while i < n1:
        arr[k] = left[i]
        i += 1
        k += 1

    while j < n2:
        arr[k] = right[j]
        j += 1
        k += 1
```

Let's explain the code!{

```python
# Lock para garantizar la exclusión mutua
lock = threading.Lock()

# merge sort function
def merge_sort(arr, low, high):
    #Base case
    if low < high:
        mid = low + (high - low) // 2

        merge_sort(arr, low, mid)
        merge_sort(arr, mid + 1, high)

        merge(arr, low, mid, high)

# thread function for multi-threading
def merge_sort_threaded(arr, low, high):
    threads = []
    section_size = MAX // THREAD_MAX

    for i in range(THREAD_MAX):
        start_index = ((THREAD_MAX + i) * section_size)
        end_index = ((THREAD_MAX + i + 1) * section_size) - 1
        t = threading.Thread(target=merge_sort, args=(arr, start_index, end_index))
        t.start()
        threads.append(t)

    # Esperar a que todos los hilos secundarios completen su trabajo
    for t in threads:
        t.join()
```

Let's explain the code!{

```python
# Save sorted list to a text file
def save_sorted_list_to_file(sorted_list):
    with open("ordenamiento.txt", "w") as file:
        for item in sorted_list:
            file.write(f"Client {item.id}: Distance from origin: {item.distance_from_origin}\n")
```

Let's explain the code!{

```python
#main
if __name__ == "__main__":
    # Conectar a la base de datos (creará una nueva si no existe)
    db_connection = sqlite3.connect('clientes.db')
    cursor = db_connection.cursor()
    delete_table(cursor)
    # Crear la tabla si no existe
    create_table(cursor)
    # Insertar ubicaciones aleatorias de clientes
    num_locations = MAX
    insert_random_locations(cursor, num_locations)
    db_connection.commit()
    print("Customer locations inserted into database.")

    # Cargar las ubicaciones de los clientes desde la base de datos
    a = load_locations_from_database(cursor)
    print("Customer locations loaded from database.")

    # Copia del arreglo para usar en merge sort secuencial
    arr_seq = [loc for loc in a]

    # t1 and t2 for calculating time for
    # merge sort multithreaded
    t1 = time.perf_counter()
    merge_sort_threaded(arr_seq, 0, len(arr_seq)-1)
    t2 = time.perf_counter()
    timeParallelism= t2-t1
    print(f"Time taken for multithreaded merge sort: {t2 - t1:.6f} seconds")
```

Let's explain the code!{

```
# t3 and t4 for calculating time for
# merge sort sequential
t3 = time.perf_counter()
merge_sort(arr_seq, 0, len(arr_seq)-1)
t4 = time.perf_counter()
timeSequential= t4-t3
print(f"Time taken for sequential merge sort: {t4 - t3:.6f} seconds")

idealParallelism = (((t4-t3)/(t2-t1))-1)*100

print("The algorithm is ",idealParallelism,"%","faster.")

speedup = timeSequential/timeParallelism
print("SpeedUp: ", speedup)

# Hay 4 procesadores
eficiencia = (speedup/ 4)*100
print(f"The algorithm is {eficiencia:.3f}% efficient.")

# Verificar si el archivo existe antes de intentar eliminarlo
if os.path.exists("ordenamiento.txt"):
    os.remove("ordenamiento.txt")

lock.acquire()
save_sorted_list_to_file(arr_seq)
lock.release()

# Cerrar la conexión con la base de datos
cursor.close()
db_connection.close()
```

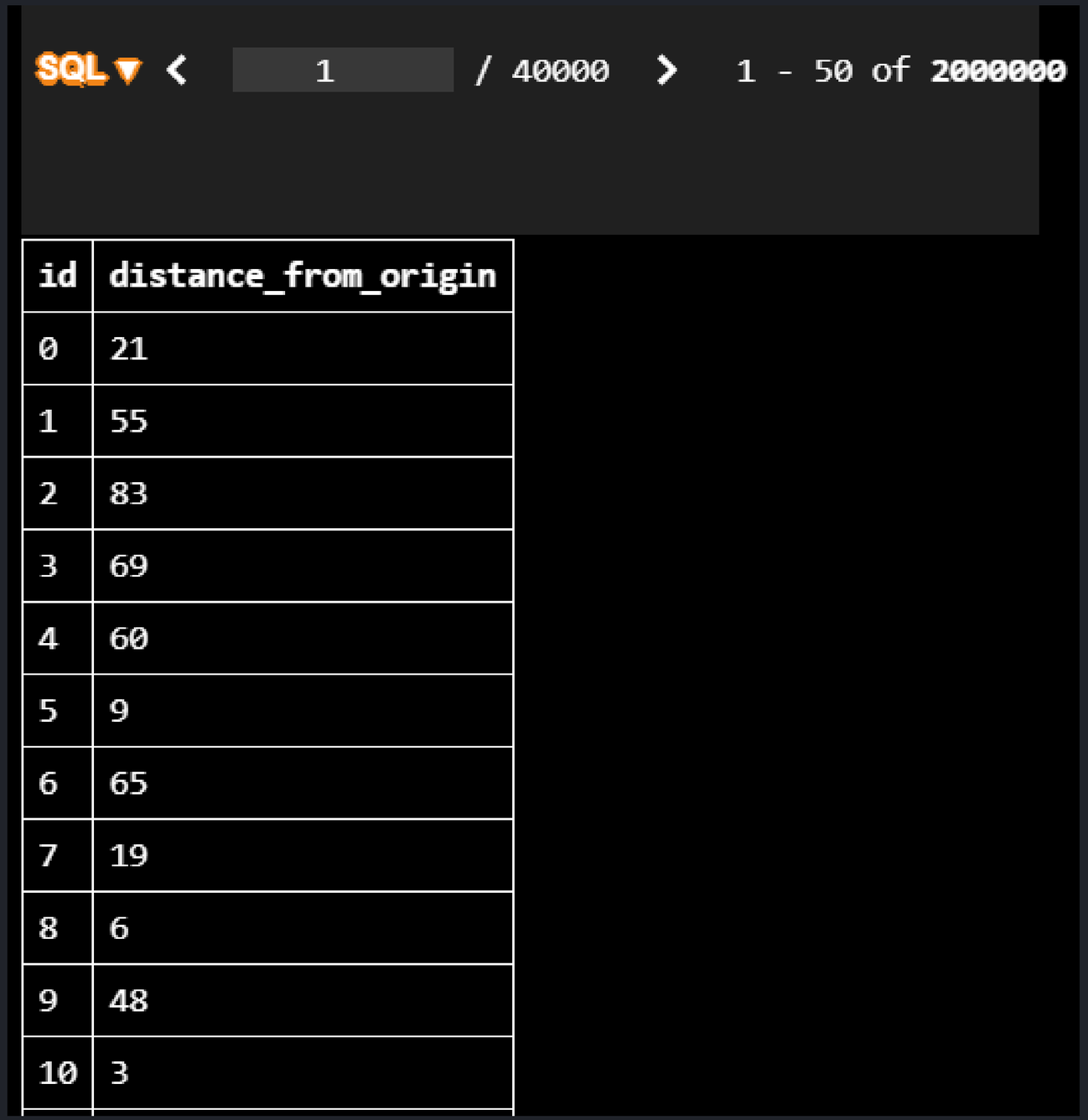
Metrics to evaluate parallel performance{

```
36    # number of elements
37    MAX = 2000000
38
39    # number of threads
40    THREAD_MAX = 8
41
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS Python

```
PS D:\ProyectoPP> & C:/Users/jatzi/AppData/Local/Microsoft/WindowsApps/python3.12.exe d:
/ProyectoPP/main.py
Customer locations inserted into database.
Customer locations loaded from database.
Time taken for multithreaded merge sort: 1.128518 seconds
Time taken for sequential merge sort: 14.969239 seconds
The algorithm is  1226.4506920265917 % faster.
SpeedUp:  13.264506920265916
The algorithm is 331.613% efficient.
PS D:\ProyectoPP>
```

Let’s check the database!



id	distance_from_origin
0	21
1	55
2	83
3	69
4	60
5	9
6	65
7	19
8	6
9	48
10	3

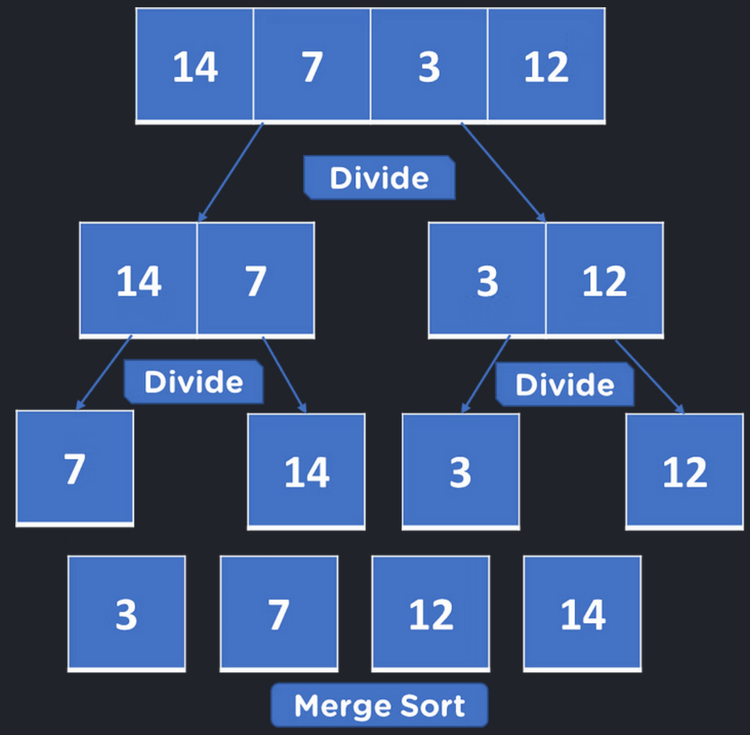
Let’s check the the sort data!

ordenamiento.txt

```
1   Client 1999990: Distance from origin: 0
2   Client 1999986: Distance from origin: 0
3   Client 1999966: Distance from origin: 0
4   Client 1999812: Distance from origin: 0
5   Client 1999649: Distance from origin: 0
6   Client 1999641: Distance from origin: 0
7   Client 1999597: Distance from origin: 0
8   Client 1999501: Distance from origin: 0
9   Client 1999382: Distance from origin: 0
10  Client 1999346: Distance from origin: 0
11  Client 1999244: Distance from origin: 0
12  Client 1999221: Distance from origin: 0
13  Client 1998905: Distance from origin: 0
14  Client 1998868: Distance from origin: 0
15  Client 1998852: Distance from origin: 0
16  Client 1998795: Distance from origin: 0
17  Client 1998780: Distance from origin: 0
18  Client 1998660: Distance from origin: 0
19  Client 1998507: Distance from origin: 0
20  Client 1998486: Distance from origin: 0
21  Client 1998395: Distance from origin: 0
22  Client 1998288: Distance from origin: 0
23  Client 1998269: Distance from origin: 0
24  Client 1998036: Distance from origin: 0
25  Client 1997932: Distance from origin: 0
26  Client 1997826: Distance from origin: 0
27  Client 1997720: Distance from origin: 0
28  Client 1997700: Distance from origin: 0
29  Client 1997679: Distance from origin: 0
30  Client 1997619: Distance from origin: 0
```

Topics{

1. Lock: It ensures only one thread accesses a shared resource at a time, preventing data corruption. Used here to manage file writing.
2. Threads: Concurrent lines of execution within a process, managed by the threading module. Employed to execute the merge sort algorithm concurrently, enhancing performance.
3. Divide and Conquer (Merge Sort): A sorting algorithm that recursively divides an array into smaller segments, sorts them independently, then merges them back together. Implemented to efficiently sort large datasets.



Thank you!

}