UNIVERSIDAD PANAMERICANA

Universidad Panamericana

Campus Guadalajara

Escuela de ingenierías

Fundamentos de programación en paralelo

REPORTE DE PROYECTO

DATOS GENERALES

Equipo		**Nombre del proyecto**	**Modelo para análisis de contagio por COVID-19**
Fecha	**21/11/2020**	**Nombre del profesor**	**Alma Nayeli Rodríguez Vázquez**
Nombre de los integrantes del equipo		**Alan Enrique Maldonado Navarro**	
		Guillermo González Mena	

OBJETIVO

El objetivo de este proyecto consiste en implementar una versión secuencial y otra en paralelo de un modelo basado en agentes para el análisis del riesgo de contagio por COVID-19 y comparar el desempeño de ambas versiones.

PROCEDIMIENTO

Realiza la implementación siguiendo estas instrucciones.

Realiza un programa en C/C++ utilizando CUDA en el que implementes el siguiente algoritmo.

Un modelo basado en agentes es utilizado para predecir el riesgo de contagio por COVID-19 y las expectativas de recuperación en los espacios de trabajo de la Universidad Panamericana. Este modelo se puede utilizar para probar estrategias de control o medidas preventivas. Con su uso, se pueden probar diferentes políticas de reapertura hipotética que son imposibles de analizar en condiciones reales.

De acuerdo con el procedimiento general del modelo basado en agentes, la descripción comienza con la fase de inicialización, seguida de la fase de operación.

Fase de inicialización

En esta fase, un conjunto A de N agentes es distribuido uniformemente en un espacio bidimensional de tamaño $p \times q$ para asignarles una posición inicial. Este espacio representa el campus de la Universidad Panamericana, mientras que los agentes representan a las personas que se encuentran en el campus.

Cada agente cuenta con un conjunto de atributos que representan diversas características individuales que influyen en la probabilidad de contagio del virus. Estas características pueden ser la edad, los hábitos de higiene, o el estado de salud de cada persona. La Tabla 1 describe los diferentes atributos que tienen los agentes y el rango de posibles valores que pueden adoptar.

En la fase de inicialización, todos los valores de los atributos de cada agente son inicializados con valores aleatorios dentro de los rangos de valores permitidos. Excepto el estatus de infección, ya que todos los agentes inician con un estatus de no infectado. Cada individuo tiene diferentes probabilidades de contagio, de contagio externo y de fatalidad. El hecho de que cada agente tenga diferentes valores dentro de los rangos establecidos simula las características particulares que cada persona tiene, como su edad y su condición de salud.

Tabla 1: Atributos de los agentes

Atributos		Rango de valores
P_{con}	Probabilidad de contagio	[0.02, 0.03]
P_{ext}	Probabilidad externa de contagio	[0.02, 0.03]
P_{fat}	Probabilidad de mortalidad	[0.007, 0.07]
P_{mov}	Probabilidad de movilidad	[0.3, 0.5]
P_{smo}	Probabilidad de movilidad en pequeñas distancias	[0.7, 0.9]
T_{inc}	Tiempo de incubación	[5, 6]
T_{rec}	Tiempo de recuperación	14
S	Estatus de infección	-No infectado (0) -Infectado (1) -En cuarentena (-1) -Fallecido (-2)
x	Posición en x	[0, p]
y	Posición en y	[0, q]

De manera similar, la probabilidad de movimiento o la probabilidad de que un agente siga las reglas y sólo se desplace a lugares cercanos de su área de trabajo es diferente para cada agente. Se asumen políticas restrictivas en las que la movilidad es limitada. Es por esto por lo que la probabilidad de movimiento es pequeña, la cual varía de 0.3 a 0.5. Además, la probabilidad de que un agente se mueva sólo a lugares esenciales es alta, por lo que varía de 0.7 a 0.9.

Además de los atributos, los parámetros de simulación descritos en la Tabla 2 deben ser inicializados.

Table 2: Valores iniciales de los parámetros de simulación

Parámetros		Valores
N	Número de agentes	10,240
d_{max}	Máximo número de días de simulación	15
M_{max}	Máximo número de movimientos por día	10
l_{max}	Radio máximo permitido para movimientos locales	5 m
R	Distancia límite de contagio	1 m
$p \times q$	Área de simulación	500 m x 500 m

Fase de operación

En la fase de operación se consideran cinco reglas que simulan los efectos del coronavirus sobre la dinámica de la población. La primera regla controla la posibilidad de transmisión del virus entre los agentes contagiados y los no contagiados. La segunda regla simula la movilidad y las interacciones entre los individuos. La tercera regla considera la probabilidad de que un agente pueda contagiarse fuera del espacio de simulación, es decir, fuera del campus. Por otro lado, la cuarta regla contempla el tiempo de incubación del virus, el tiempo que tarda un agente en presentar síntomas, el tiempo de cuarentena y de recuperación. Finalmente, la quinta regla simula los desafortunados casos fatales. Estas reglas representan los principales componentes para modelar el comportamiento del COVID-19 dentro de un área de interés. Una descripción más detallada de cada regla se muestra a continuación.

Regla 1: Contagio

Con base en el último informe por parte de la Organización Mundial de la Salud acerca del COVID-19, las medidas de distanciamiento tienen como objetivo frenar la propagación de la enfermedad. Por tanto, la distancia física recomendada es de al menos un metro. Sin embargo, en los espacios de trabajo, la proximidad puede ser indispensable para la comunicación entre compañeros. Esta necesidad de comunicación supone una posible exposición al virus. Por tanto, esta regla simula el riesgo de contagio cuando los individuos interactúan entre sí descuidando la sana distancia.

En tales circunstancias, la regla uno considera que cada agente tiene una probabilidad P_{con} de infectarse, la cual depende de su condición de salud particular, edad y hábitos de higiene como lavarse las manos con frecuencia, etiqueta al toser y uso de mascarillas. Por lo tanto, si un individuo no infectado a_i está rodeado por al menos un vecino infectado a_j, y el agente vecino a_j está dentro del radio de contagio R, entonces el individuo a_i es propenso a infectarse por a_j con una probabilidad P_{con}. Esta regla se expresa como:

$$S^d(a_i) = (rand(0,1) \leq P_{con}(a_i))\,\alpha \qquad (2)$$

En donde S^d es el estatus de contagio actual para el agente a_i en el día d. Un valor aleatorio en el intervalo [0,1] es representado por $rand(0,1)$. El término α es la función de activación que advierte un posible riesgo de contagio si un individuo infectado está muy cerca de a_i, el cual está definido como:

$$\alpha = \{1,\ \left(\sum_{j=1,\forall j \neq i}^{N} \left(dist\left(a_i, a_j\right) \leq R \right) \beta \right) \geq 1\ 0,\ \ en\ otro\ caso \qquad (3)$$

En donde $dist\left(a_i, a_j\right)$ es la distancia Euclidiana entre el agente a_i y a_j. Por otro lado, β es la función de activación que determina si el agente a_j está infectado. Su definición está dada como:

$$\beta = \{1,\ \ S^d\left(a_j\right) > 0\ 0,\ \ en\ otro\ caso \qquad (4)$$

Estas definiciones constituyen la regla uno, en la cual el estatus actual S^d de un agente a_i puede cambiar a infectado con una probabilidad P_{con} si un individuo a_j está a menos de un metro de distancia. De lo contrario, el estatus del agente se mantiene sin cambios.

Regla 2: Movilidad

La movilidad es esencial en los espacios de trabajo, pero es una de las principales causas de contagio. No se puede restringir en absoluto, pero se puede controlar de alguna manera. Sin embargo, incluso imponiendo medidas restrictivas, debe tenerse en cuenta que, en ocasiones, las personas pueden ignorar esas restricciones. Por lo tanto, el modelo considera dos posibilidades principales en cuanto al movimiento de agentes: el desplazamiento a lugares cercanos (siguiendo las medidas restrictivas) y lugares distantes (ignorando las reglas restrictivas).

Los lugares cercanos comprenden aquellos espacios indispensables que satisfacen las necesidades básicas del ser humano como el baño, el dispensador de agua, las aulas, el área de café, por mencionar algunos. Por otro lado, posibles lugares lejanos pueden incluir otros departamentos, la biblioteca, áreas comunes de recreación, restaurantes y cafeterías dentro de las instalaciones.

En el modelo, cada agente tiene una probabilidad de movimiento P_{mov} que imita la decisión personal de trasladarse a otro lugar. Según esta probabilidad, un agente puede trasladarse a otra ubicación o permanecer en su posición actual. Además, cada agente tiene una probabilidad de movimiento P_{smo}, que simula la decisión particular de trasladarse a un lugar cercano o lejano.

La posición en los ejes x y y de los agentes puede cambiar de acuerdo con la siguiente expresión:

$$x^{d+1}(a_i) = \{X(a_i),\ \gamma \neq 0\ x^d(a_i),\ en\ otro\ caso \tag{5}$$

$$y^{d+1}(a_i) = \{Y(a_i),\ \gamma \neq 0\ y^d(a_i),\ en\ otro\ caso \tag{6}$$

En donde x^{d+1} y y^{d+1} son las nuevas posiciones del agente a_i. La posición actual del agente a_i está determinado por x^d y y^d, mientras que γ es la función de activación que determina si un agente ha decidido moverse o mantenerse en el mismo lugar, la cual está definida como:

$$\gamma = \{1,\ rand(0,1) \leq P_{mov}(a_i)\ 0,\ otherwise \tag{7}$$

Los términos $X(a_i)$ y $Y(a_i)$ de las ecuaciones (5) y (6) determinan el movimiento de los agentes en los ejes x y y, respectivamente. Si el agente decide moverse, la siguiente decisión será el tipo de movimiento, el cual puede ser a un lugar cercano o distante. Ambos tipos de movimiento para $X(a_i)$ y $Y(a_i)$ pueden formularse con (8) y (9), respectivamente:

$$X(a_i) = (x^d(a_i) + (2(rand(0,1)) - 1)\, l_{max})\, \delta + p(rand(0,1))(1-\delta) \tag{8}$$

$$Y(a_i) = (y^d(a_i) + (2(rand(0,1)) - 1)\, l_{max})\, \delta + q(rand(0,1))(1-\delta) \tag{9}$$

En donde X y Y corresponden la nueva posición del agente a_i en el eje x y y, respectivamente. De la ecuación (8), el primer término $(x^d(a_i) + (2(rand(0,1)) - 1)\, l_{max})\, \delta$ simula un pequeño movimiento, en donde l_{max} es el máximo radio para movimientos locales. Por otro lado, el segundo término $p(rand(0,1))(1-\delta)$ simula un desplazamiento a un lugar lejano, en donde p corresponde al ancho del espacio de simulación. De manera similar, el primer término de la ecuación (9) corresponde a un movimiento local, mientras que el

segundo término a un movimiento lejano, en el que q es el largo del espacio de simulación. La función de activación δ determina si un agente ha decidido hacer un movimiento a un lugar cercano o no, y su definición está dada por:

$$\delta = \{1, \quad rand(0,1) \leq P_{smo}(a_i)\ 0, \quad otherwise \tag{10}$$

De la ecuación (10), si el valor de δ es uno, el segundo término de las ecuaciones (8) y (9) se vuelve cero, lo cual origina un movimiento local. Por otro lado, si el valor de δ es cero, entonces el primer término de las ecuaciones (8) y (9) se vuelve cero, ocasionando un movimiento largo.

Regla 3: Contagio externo

Esta regla simula la posibilidad de que una persona se infecte fuera del espacio de trabajo. Aunque el modelo basado en agentes simula la propagación del COVID-19 en un entorno específico, no sería realista si no se considerara la probabilidad de contagio externo. Por tanto, esta regla simula lo que ocurre más allá de las instalaciones de forma general.

Después de una jornada laboral normal, las personas están expuestas a infecciones en otros lugares, ya que pueden tener contacto con personas infectadas fuera del campus. En consecuencia, en el modelo se considera la probabilidad de contagio externo, donde los agentes pueden iniciar una nueva jornada laboral ya infectados, poniendo en riesgo la salud de sus compañeros.

Según la Organización Mundial de la Salud, el inicio de los síntomas por COVID-19 no es inmediato, esto significa que, si un individuo se infecta, aún no se puede detectar. La enfermedad tarda un tiempo específico en manifestar síntomas. Este tiempo se denomina tiempo de incubación, que varía en promedio de 5 a 6 días. Por lo tanto, incluso si hay filtros sanitarios instalados en la entrada de las instalaciones para detectar síntomas relacionados con COVID-19, un individuo infectado puede pasar desapercibido.

Bajo tales consideraciones, la regla tres considera que un individuo no infectado a_i puede ser infectado (cambiando su estatus) de acuerdo con una probabilidad de infección externa P_{ext} expresada como:

$$S^{d+1}(a_i) = \{1, \quad (rand(0,1) \leq P_{ext}(a_i))\ \varepsilon > 0\ S^d(a_i), \quad en\ otro\ caso \tag{11}$$

En donde la función de activación ε indica si un agente a_i puede infectarse externamente:

$$\varepsilon = \{0, \quad S^d(a_i) \neq 0\ 1, \quad otherwise \tag{12}$$

Regla 4: Tiempo de incubación, inicio de síntomas, cuarentena y tiempo de recuperación

Una vez que una persona ha estado expuesta al coronavirus, la regla cuatro simula el proceso durante el tiempo de incubación, el inicio de los síntomas, el inicio de la cuarentena y la recuperación final. Se puede detectar un individuo infectado después del tiempo de incubación cuando los síntomas asociados con el COVID-19 son evidentes. Si los síntomas no son graves, la recomendación es el aislamiento domiciliario, que corresponde al inicio de la cuarentena. En un escenario favorable, después del período de cuarentena, se espera una recuperación.

La información proporcionada por la Organización Mundial de la Salud indica que el tiempo de incubación varía en promedio de 5 a 6 días. Después de eso, las personas experimentan síntomas durante 14 días, que se considera el período de recuperación en cuarentena para síntomas leves / moderados. Hasta ahora, no hay evidencia que respalde que una persona recuperada pueda volver a infectarse. Por lo tanto, una vez finalizada la cuarentena, el individuo recuperado no puede infectar a otras personas.

Según la información presentada, la regla cuatro realiza un seguimiento de los días transcurridos durante el tiempo de incubación T_{inc} de cada agente infectado a_i. Por lo tanto, el tiempo de incubación se actualiza todos los días d. Cuando el tiempo de incubación se termina, (después de 5 o 6 días dependiendo de cada individuo) los síntomas son evidentes, por lo que el estatus del agente infectado cambia a en cuarentena. Una vez que el individuo está en cuarentena, la regla cuatro registra los días trascurridos después de que se han presentado los síntomas para simular el tiempo de recuperación T_{rec}. Después de 14 días, la cuarentena termina. Entonces, el modelo considera que los agentes recuperados ya no pueden contagiarse ni contagiar a otros.

Basado en esta información, la regla cuatro lleva el conteo de los días transcurridos durante el tiempo de incubación T_{inc} de cada agente contagiado a_i. Por lo tanto, el tiempo de incubación es actualizado cada día d como sigue:

$$T_{inc}^{d+1}(a_i) = \{T_{inc}^{d}(a_i) - 1, \;\; S^{d}(a_i) > 0 \; T_{inc}^{d}(a_i), \;\; en\ otro\ caso \tag{13}$$

Cuando el tiempo de incubación termina (después de 5 o 6 días dependiendo de cada individuo), los síntomas se hacen evidentes, por lo que el estatus del agente infectado cambia a en cuarentena. La actualización del estatus se expresa como:

$$S^{d+1}(a_i) = \{S^{d}(a_i), \;\; T_{inc}^{d}(a_i) > 0 \;\; -1, \;\; en\ otro\ caso \tag{14}$$

Además, una vez que un agente está en cuarentena, la regla cuatro también registra los días transcurridos después de que los síntomas se hicieron evidentes para simular el tiempo de recuperación T_{rec}, el cual es calculado como:

$$T_{rec}^{d+1}(a_i) = \{T_{rec}^{d}(a_i) - 1, \;\; S^{d}(a_i) < 0 \; T_{rec}^{d}(a_i), \;\; en\ otro\ caso \tag{15}$$

Después de 14 días transcurridos, la cuarentena termina. Entonces, el modelo considera que los agentes recuperados no pueden contagiarse nuevamente ni pueden contagiar a otros agentes.

$$S^{d+1}(a_i) = \{S^{d}(a_i), \;\; T_{rec}^{d}(a_i) > 0 \; isCured, \;\; en\ otro\ caso$$

Regla 5: Casos fatales

Desafortunadamente, el COVID-19 ha dejado considerables casos fatales en todo el mundo. Según la información recopilada en todo el mundo desde diciembre de 2019 hasta mayo de 2020, la tasa bruta global de casos letales (CFR), que es la proporción de episodios mortales de enfermedad, es del 7%. Estos casos corresponden a aquellas personas infectadas que experimentaron síntomas graves debido a

varias condiciones médicas subyacentes como diabetes, hipertensión, enfermedad respiratoria crónica, enfermedad cardiovascular y cáncer. El riesgo de enfermedad grave también aumenta con la edad. Además, el 20% de los casos requiere hospitalización, mientras que el 5% requiere cuidados intensivos y ventilación, impactando en los sistemas de salud.

Con base en la información presentada sobre casos fatales, el modelo incluye una probabilidad de mortalidad P_{fat} para cada agente a_i. Según el CFR, esta probabilidad varía de 0 a 0.07. Por lo tanto, si un individuo a_i está en cuarentena, entonces este agente puede morir cualquier día después del inicio de los síntomas, dependiendo de su CFR. Cuando un agente muere, su estatus cambia. Por lo tanto, la regla cinco simula los desafortunados casos fatales utilizando la siguiente expresión:

$$S^{d+1}(a_i) = \{-2, \quad \left(rand(0,1) \leq P_{fat}(a_i)\right) \sigma > 0 \; S^d(a_i), \quad en\ otro\ caso \tag{16}$$

En donde la función de activación σ indica si un agente a_i está en el periodo de recuperación y se expresa de la siguiente forma:

$$\sigma = \{1, \quad S^d(a_i) < 0 \; 0, \quad en\ otro\ caso \tag{17}$$

Procedimiento general

El modelo basado en agentes deberá ser simulado computacionalmente para analizar el comportamiento del COVID-19 utilizando las cinco reglas presentadas. El procedimiento de simulación comienza con la fase de inicialización, en la que se deben configurar los parámetros generales y los atributos de los agentes. Después de eso, comienza la fase de operación, que se ejecuta iterativamente hasta que se alcanza el número máximo de iteraciones d_{max}.

Cada iteración se considera un día de simulación, mientras que un día de simulación se divide en dos: el tiempo que las personas pasan dentro del campus y el que pasan fuera del campus. Además, se supone que, en cada día de simulación, los agentes pueden realizar varios movimientos pequeños o distantes en el espacio de trabajo, lo que imita la actividad habitual de una persona en un día laboral/escolar típico. Si un individuo cambia de posición como consecuencia de la regla dos, entonces está expuesto a contagio por la regla uno. Por tanto, un movimiento comprende la aplicación de las reglas uno y dos para todos los agentes.

El número máximo de movimientos por día de simulación está determinado por M_{max}. Así, se completa una jornada laboral/escolar de simulación cuando se alcanza el número máximo de movimientos por día. Posteriormente, se considera que las personas se encuentran fuera de las instalaciones. Por lo tanto, la infección externa se efectúa según la regla tres. Finalmente, un día de simulación termina cuando se aplican las reglas cuatro y cinco.

El procedimiento general se resume en el algoritmo 1.

Algoritmo 1: Procedimiento computacional del modelo basado en agentes

Fase de inicialización:

| Configurar los atributos de los agentes

| Configurar los parámetros de simulación

Fin de la fase de inicialización

Fase de operación:

| Mientras que el día actual de simulación sea menor que d_{max}
| | Mientras que el movimiento actual sea menor que M_{max}
| | | Para todos los agentes
| | | | Aplicar regla uno
| | | | Aplicar regla dos
| | | Fin del ciclo
| | Fin del ciclo
| | Para todos los agentes
| | | Aplicar la regla tres
| | | Aplicar la regla cuatro
| | | aplicar la regla cinco
| | Fin del ciclo
| Fin del ciclo
Fin de la fase de operación
Reportar los resultados

Resultados
Los resultados del proceso de simulación deberán reportar los siguientes índices:

- Número de casos acumulados de agentes contagiados
- Número de nuevos casos positivos por día
- Número de casos acumulados de agentes recuperados
- Número de casos recuperados por día
- Número de casos fatales acumulados
- Número de casos fatales por día
- El día en que se contagió el primer agente, el 50% y el 100% de la población.
- El día en que se recuperó el primer agente, la mitad, y la totalidad de los agentes recuperados.
- El día en ocurrió el primer caso fatal, la mitad de los casos fatales y la totalidad de los casos fatales.
- Tiempo de ejecución del modelo en el CPU y en el GPU

IMPLEMENTACIÓN

Agrega el código de tu implementación aquí.

GPU:

- ***classes.cpp***

```
#include <vector>


class Agent
{
public:
    // Position
    int id = 0;
    int x = 0;
    int y = 0;

    bool hasBeenAccountedFor = false;
```

```
    bool hasDied = false;

    // No infectado = 0
    // Infectado = 1
    // En Cuarentena = -1
    // Muerto = -2
    int S = 0;
    // Probabilidad de Contagio
    float Pcon = 0;
    // Probabilidad de Contagio Externa
    float Pext = 0;
    //Probabilidad mortalidad
    float Pfat = 0;
    // Probabilidad de movilidad
    float Pmov = 0;
    //Probabilidad de movilidad en peque□as distancias
    float Psmo = 0;
    //Tiempo de incubaci□n
    int Tinc = 0;
    //Tiempo de recuperaci□n
    int Trec = 14;
    //Constructor
    Agent(int x, int y, int i);
    Agent() = default;
    //Rango de probabilidad de contagio
    static float PconRange[2];
    //Rango de probabilidad externa de contagio
    static float PextRange[2];
    //Rango de probabilidad de mortalidad
    static float PfatRange[2];
    // Rango de probabilidad de movilidad
    static float PmovRange[2];
    // Rango de probabilidad de Movimiento en peque□a distancia
    static float PsmoRange[2];
    //Rango de tiempo de incubaci□n
    static int TincRange[2];
};

class GlobalState
{
```

```
public:
    //N◆mero de agentes
    static int N;
    //M◆ximo n◆mero de d◆as de simulaci◆n
    static int Dmax;
    //M◆ximo n◆mero de movimientos por d◆a
    static int Mmax;
    //Radio m◆ximo permitido para movimientos locales
    static int lmax;
    //Distancia l◆mite de contagio
    static int R;
    //◆rea de simulaci◆n, Dimensi◆n 0:y, Dimensi◆n 1:x
    static int mapSize[2];

    /* Estadisticas */
    static int infectedAgents;
    static int curedAgents;
    static int deadAgents;
    static int patientZeroDay;
    static int halfPopulationInfectedDay;
    static int fullPopulationInfectedDay;
    static int patientZeroCuredDay;
    static int halfPopulationCuredDay;
    static int fullPopulationCuredDay;
    static int patientZeroDeadDay;
    static int halfPopulationDeadDay;
    static int fullPopulationDeadDay;
};

class Operations
{
public:
    //First Operation
    static int hasBeenInfected(Agent* agent, std::vector<Agent*>* AgentList);
    //Second Operation
    static int isShortMovement(Agent* agent);
    static int XMovement(Agent* agent);
    static int YMovement(Agent* agent);
    static bool willMove(Agent* agent);
    //Third Operation
```

```
    static int hasBeenExternalInfected(Agent* agent);
    //Fourth Operation
    static int incubationTime(Agent* agent);
    static int hasSymptoms(Agent* agent);
    static int recuperationTime(Agent* agent);
    //Fifth Operation
    static int isInRecuperation(Agent* agent);
    static int isDead(Agent* agent);
    //Helpers
    static float GenerateRandomFloatBetween(float a, float b);
    static int GenerateRandomIntegerBetween(int a, int b);
    static int Clamp(int value, int low, int high);
};
```

- ***kernel.cu***

```
#include <stdio.h>
#include <stdlib.h>
#include <iostream>
#include <cstring>
#include <ctime>
#include <math.h>
#include <time.h>
#include <random>
#include <chrono>
#include <vector>
#include <string>
#include <sstream>

#include "classes.cpp"

/* CUDA Runtime */
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include <curand_kernel.h>

__host__ void check_CUDA_error(const char* msg) {
    cudaError_t err;
    cudaDeviceSynchronize();
    err = cudaGetLastError();
    if (err != cudaSuccess) {
        printf("Error (%d): [%s] %s\n", err, msg, cudaGetErrorString(err));
```

```
    }
}

using namespace std;

class Agent;
class Operations;
class GlobalState;

typedef std::chrono::high_resolution_clock myclock;
myclock::time_point beginning = myclock::now();
myclock::duration d = myclock::now() - beginning;
unsigned seed2 = d.count();
std::mt19937 rng(seed2);
std::default_random_engine generator;

__global__ void randomCudaGenerator(int min, int max, double* result)
{
    int tId = threadIdx.x + (blockIdx.x * blockDim.x);
    curandState state;
    curand_init((unsigned long long)clock() + tId, 0, 0, &state);

    double rand1 = curand_uniform_double(&state) * (max - min) + min;
    *result = rand1;
    printf("randomCudaGenerator :: %f\n", rand1);
}

 float GenerateRandomFloatBetween(float a, float b)
{
    float res;
    std::uniform_real_distribution<double> distribution(a, b);
    res = distribution(generator);

    return res;
}
 int GenerateRandomIntegerBetween(int a, int b)
{
    std::uniform_int_distribution<int> gen(a, b);
    return gen(rng);
}
```

```
/*Operations Declarations*/
//First Operation
int hasBeenInfected(Agent* agent, vector<Agent*>* AgentList)
{
    if (agent->S != 0) {
        return agent->S;
    }
    int res = 0;
    // Part First Part
    float prob = GenerateRandomFloatBetween(0, 1);
    bool fp = prob <= agent->Pcon ? 1 : 0;
    bool a = 0;

    // Part: Calculate Alpha
    bool sp = 0;
    bool tp = 0;
    int validateFn = 0;
    int sum = 0;

    for (Agent* agent_test : *AgentList)
    {
        if (agent->x == agent_test->x && agent->y == agent_test->y)
        {
            continue;
        }

        float distance = (float)sqrt(pow(agent_test->x - agent->x, 2) + pow(agent_test->y
- agent->y, 2));
        sp = distance <= GlobalState::R;
        // Part: Calculate Beta
        tp = agent_test->S > 0 ? 1 : 0;
        sum += sp * tp;
    }
    a = sum >= 1 ? 1 : 0;


    return (fp * a) ? 1 : agent->S;
}
```

```
//Second Operation
int isShortMovement(Agent* agent)
{
    float prob = GenerateRandomFloatBetween(0, 1);
    return (prob <= agent->Psmo) ? 1 : 0;
}

int XMovement(Agent* agent)
{
    int s = isShortMovement(agent);
    int p = GlobalState::mapSize[1];
    float prob = GenerateRandomFloatBetween(0, 1);
    float longDistance = p * prob * (1 - s);
    prob = GenerateRandomFloatBetween(0, 1);
    float shortDistance = (agent->x + (2 * prob - 1) * GlobalState::lmax) * s;
    return shortDistance + longDistance;
}

int YMovement(Agent* agent)
{
    int s = isShortMovement(agent);
    int q = GlobalState::mapSize[0];
    float prob = GenerateRandomFloatBetween(0, 1);
    float longDistance = q * prob * (1 - s);
    prob = GenerateRandomFloatBetween(0, 1);
    float shortDistance = (agent->y + (2 * prob - 1) * GlobalState::lmax) * s;
    return shortDistance + longDistance;
}

bool willMove(Agent* agent)
{
    float prob = GenerateRandomFloatBetween(0, 1);
    return (prob <= agent->Pmov) ? 1 : 0;
}

//Third Operation
int hasBeenExternalInfected(Agent* agent)
{
    float prob = GenerateRandomFloatBetween(0, 1);
    //Part: Calculate First Part
```

```
    bool fp = prob <= agent->Pext ? 1 : 0;
    //Part: Calcula Epsilon
    bool sp = agent->S != 0 ? 0 : 1;
    //fp * sp == if fp or sp are 0 the condition is false
    bool finalB = (fp * sp) > 0;

    return (finalB) ? 1 : agent->S;
}

//Third Operation
__device__ int hasBeenExternalInfected_GPU(Agent* agent, float prob)
{
    //Part: Calculate First Part
    bool fp = prob <= agent->Pext ? 1 : 0;
    //Part: Calcula Epsilon
    bool sp = agent->S != 0 ? 0 : 1;
    //fp * sp == if fp or sp are 0 the condition is false
    bool finalB = (fp * sp) > 0;

    return (finalB) ? 1 : agent->S;
}

//Fourth Operation
__device__ int incubationTime(Agent* agent)
{
    int Tinc = agent->Tinc;
    if (agent->S > 0)
    {
        Tinc--;
    }
    return Tinc;
}

__device__ int hasSymptoms(Agent* agent)
{
    int S = -1;
    if (agent->Tinc > 0)
    {
        S = agent->S;
    }
```

```
    return S;
}

__device__ int recuperationTime(Agent* agent)
{
    int Trec = agent->Trec;
    if (agent->S == -1)
    {
        Trec--;
    }
    return Trec;
}

//Fifth Operation
 int isInRecuperation(Agent* agent)
{
    return (agent->S < 0) ? 1 : 0;
}
__device__ int isInRecuperation_GPU(Agent* agent)
{
    return (agent->S < 0) ? 1 : 0;
}

int isDead(Agent* agent)
{
    int o = isInRecuperation(agent);
    float prob = GenerateRandomFloatBetween(0, 1);
    int Pfat = (prob <= agent->Pfat) ? 1 : 0;
    return (Pfat * o > 0) ? -2 : agent->S;
}

__device__ int isDead_GPU(Agent* agent, float prob)
{
    int o = isInRecuperation_GPU(agent);
    //float prob = GenerateRandomFloatBetween(0, 1);
    int Pfat = (prob <= agent->Pfat) ? 1 : 0;
    return (Pfat * o > 0) ? -2 : agent->S;
}

int Clamp(int value, int low, int high)
```

```
{
    return value > high ? high : value < low ? low : value;
}

/*GlobalState Declarations*/
int GlobalState::mapSize[2] = { 500, 500 };
int GlobalState::Dmax = 60; //// DIAS
int GlobalState::Mmax = 10;
int GlobalState::lmax = 5;
int GlobalState::N = 10240; //// AGENTES
// int GlobalState::N = 5000;
int GlobalState::infectedAgents = 0;
int GlobalState::R = 1;
/* Statistics*/
int GlobalState::curedAgents = 0;
int GlobalState::deadAgents = 0;
int GlobalState::patientZeroDay = 0;
int GlobalState::halfPopulationInfectedDay = 0;
int GlobalState::fullPopulationInfectedDay = 0;
int GlobalState::patientZeroCuredDay = 0;
int GlobalState::halfPopulationCuredDay = 0;
int GlobalState::fullPopulationCuredDay = 0;
int GlobalState::patientZeroDeadDay = 0;
int GlobalState::halfPopulationDeadDay = 0;
int GlobalState::fullPopulationDeadDay = 0;

float Agent::PconRange[2] = { 0.02, 0.03 };
float Agent::PextRange[2] = { 0.02, 0.03 };
float Agent::PfatRange[2] = { 0.007, 0.07 };
float Agent::PmovRange[2] = { 0.3, 0.5 };
float Agent::PsmoRange[2] = { 0.7, 0.9 };
int Agent::TincRange[2] = { 5, 6 };

/*Agent Constructor*/
Agent::Agent(int x, int y, int i)
{
    // Position
    Agent::id = i;
    Agent::x = x;
    Agent::y = y;
```

```
    // Assing the Infection Probability
    Agent::Pcon = GenerateRandomFloatBetween(Agent::PconRange[0], Agent::PconRange[1]);
    // Assign the External Infection Probability
    Agent::Pext = GenerateRandomFloatBetween(Agent::PextRange[0], Agent::PextRange[1]);
    //Assign the Mortality Probability
    Agent::Pfat = GenerateRandomFloatBetween(Agent::PfatRange[0], Agent::PfatRange[1]);
    // Assign the Movement Probability
    Agent::Pmov = GenerateRandomFloatBetween(Agent::PmovRange[0], Agent::PmovRange[1]);
    // Assign the Small Movement Probability
    Agent::Psmo = GenerateRandomFloatBetween(Agent::PsmoRange[0], Agent::PsmoRange[1]);
    // Assign the Incubation Time
    Agent::Tinc = GenerateRandomFloatBetween(Agent::TincRange[0], Agent::TincRange[1]);
};

__global__ void Rule345(Agent* AgentList) {

    int min_1 = 0;
    int max_1 = 1;
    int min_2 = 0;
    int max_2 = 1;
    curandState state;
    int tId = threadIdx.x + (blockIdx.x * blockDim.x);
    curand_init((unsigned long long)clock() + tId, 0, 0, &state);
    double rand1 = curand_uniform_double(&state) * (max_1 - min_1) + min_1;
    double rand2 = curand_uniform_double(&state) * (max_2 - min_2) + min_2;
    int gId = blockIdx.x * blockDim.x + threadIdx.x;

    Agent agent = AgentList[gId];
    int previousState = agent.S;

    //Aplicar Regla 3
    agent.S = hasBeenExternalInfected_GPU(&agent, (float)rand1);

    //Aplicar Regla 4
    agent.Tinc = incubationTime(&agent);
    agent.S = hasSymptoms(&agent);
    agent.Trec = recuperationTime(&agent);
    if (agent.Trec <= 0) {
        agent.Pcon = -1;
        agent.Pext = -1;
```

```
        agent.Pfat = -1;
        agent.S = 0;

        if (!agent.hasBeenAccountedFor) {
            agent.hasBeenAccountedFor = true;
        }

    }
    //Aplicar Regla 5
    agent.S = isDead_GPU(&agent, (float)rand2);

    AgentList[gId] = agent;
}


/**/
int main()
{
    clock_t total_start_CPU = clock();

    /* 1. Map Generation and Setup */
    int size0 = 500;
    int size1 = 500;
    vector<int> historyInfectedPerDay;
    vector<int> historyCuredPerDay;
    vector<int> historyDeadPerDay;

    Agent* AgentList = new Agent[GlobalState::N];
    Agent **AgentMap = new Agent*[500 * 500];


        for (int i = 0; i < size0; i++)
        {
            for (int j = 0; j < size1; j++)
            {
                AgentMap[i * 500 + j] = NULL;
            }
        }
```

```
/* 2. Generate Agents on Map */
int leftAgents = GlobalState::N;
int i = 0;
while (leftAgents > 0)
{
    int x = GenerateRandomIntegerBetween(0, 499);
    int y = GenerateRandomIntegerBetween(0, 499);

    if (AgentMap[y * 500 + x] != NULL)
    {
        continue;
    }

    Agent* agent = new Agent(x, y, i);
    AgentMap[y * 500 + x] = agent;

    AgentList[i] = *agent;
    leftAgents--;
    i++;
}

/* Operation Phase*/
int currentDay = 0;
//Mientras que el d�a actual de simulaci�n sea menor que dMax
while (currentDay < GlobalState::Dmax)
{
    int infectedToday = 0;
    int killedToday = 0;
    int curedToday = 0;

    printf("== Day %d (%d)\n ", (currentDay + 1), GlobalState::infectedAgents);
    int currentMovement = 0;
    //Mientras que el movimiento actual sea menor que mMax
    while (currentMovement < GlobalState::Mmax)
    {
        //Para todos los agentes
        for (int i = 0; i < GlobalState::N; i++)
        {
            Agent* agent = &AgentList[i];
            int previousState = agent->S;
```

```
                //Aplicar Regla 1
                vector<Agent*> Neighbours;
                if (agent->y > 0 && AgentMap[(agent->y - 1)*500+(agent->x)] != NULL)
                {
                    Neighbours.push_back(AgentMap[(agent->y - 1) * 500 + (agent->x)]);
                }
                if (agent->x > 0 && AgentMap[(agent->y)*500 + (agent->x - 1)] != NULL)
                {
                    Neighbours.push_back(AgentMap[(agent->y) * 500 + (agent->x - 1)]);
                }
                if (agent->x < size0 - 1 && AgentMap[(agent->y) * 500 + (agent->x + 1)] !=
NULL)
                {
                    Neighbours.push_back(AgentMap[(agent->y) * 500 + (agent->x + 1)]);
                }
                if (agent->y < size1 - 1 && AgentMap[(agent->y + 1)*500 + (agent->x)] !=
NULL)
                {
                    Neighbours.push_back(AgentMap[(agent->y + 1) * 500 + (agent->x)]);
                }
                agent->S = hasBeenInfected(agent, &Neighbours);


                //Aplicar Regla 2
                if (willMove(agent))
                {
                    int x;
                    int y;

                    do
                    {
                        x = XMovement(agent);
                        y = YMovement(agent);

                        x = Clamp(x, 0, size0 - 1);
                        y = Clamp(y, 0, size1 - 1);
                    } while (AgentMap[y * 500 + x] != NULL);

                    //Se elimina de la vieja posici□n
```

```
                    AgentMap[agent->y * 500 + agent->x] = NULL;

                    //Se a□ade a la nueva posici□n
                    agent->x = x;
                    agent->y = y;
                    AgentMap[y * 500 + x] = agent;
                }

                if (agent->S != previousState) {
                    switch (agent->S) {
                    case 1: //Infected
                        if (GlobalState::infectedAgents == 0)
                        {
                            printf("\t\tPatient Zero has been found\n");
                            GlobalState::patientZeroDay = currentDay;
                        }
                        GlobalState::infectedAgents++;
                        infectedToday++;
                        break;
                    }
                }
            }

            currentMovement++;
        }

        /* KERNEL 345 */

        Agent* DEV_AgentList;
        Agent* DEV_Result_AgentList;
        DEV_Result_AgentList = (Agent*)malloc(GlobalState::N * sizeof(Agent));

        cudaMalloc((void**)&DEV_AgentList, GlobalState::N * sizeof(Agent));
        check_CUDA_error("Malloc DEV_AgentList :: 345");

        cudaMemcpy(DEV_AgentList, AgentList, GlobalState::N * sizeof(Agent),
cudaMemcpyHostToDevice);
        check_CUDA_error("Memcpy Data HOST :: DEV 345");

        dim3 grid(10);
```

```
        dim3 block(1024);

        Rule345 << <grid, block >> > (DEV_AgentList);
        check_CUDA_error("Rule345 ::");

        cudaMemcpy(DEV_Result_AgentList, DEV_AgentList, GlobalState::N * sizeof(Agent),
cudaMemcpyDeviceToHost);
        check_CUDA_error("Memcpy Data DEV :: HOST 345");

        int infectedAgentsGPU = 0;
        int curedAgentsGPU = 0;
        int deadAgentsGPU = 0;

        int totalInfected = 0;

        for (int i = 0; i < GlobalState::N; i++) {
            Agent* gpu_agent = &DEV_Result_AgentList[i];

            switch (gpu_agent->S)
            {
            case 1:
                //Infected

                infectedAgentsGPU++;
                totalInfected++;
                break;
            case -1:
                totalInfected++;
                break;
            case -2:
                //Dead
                gpu_agent->Pcon = -1;
                gpu_agent->Pext = -1;
                gpu_agent->Pfat = -1;
                gpu_agent->S = 0;
                if (!gpu_agent->hasBeenAccountedFor) {
                    deadAgentsGPU++;
                    gpu_agent->hasBeenAccountedFor = true;
                    gpu_agent->hasDied = true;
                }
```

```
            totalInfected++;
            break;
        case 0:
            if (gpu_agent->hasBeenAccountedFor && !gpu_agent->hasDied) {
                curedAgentsGPU++;
                totalInfected++;
            }
            break;
        }
    }

    infectedToday += totalInfected - GlobalState::infectedAgents;
    if (infectedToday < 0) {
        infectedToday = 0;
    }
    killedToday += deadAgentsGPU - GlobalState::deadAgents;
    if (killedToday < 0) {
        killedToday = 0;
    }
    curedToday += curedAgentsGPU - GlobalState::curedAgents;
    if (curedToday < 0) {
        curedToday = 0;
    }

    historyInfectedPerDay.push_back(infectedToday);
    historyDeadPerDay.push_back(killedToday);
    historyCuredPerDay.push_back(curedToday);

    printf("\tInfected: %d\n", infectedToday);
    printf("\tDead: %d\n", killedToday);
    printf("\tCured: %d\n", curedToday);


    int sumInfected = 0;
    int sumKilled = 0;
    int sumCured = 0;
    for (int i = 0; i < historyInfectedPerDay.size(); i++) {
        sumInfected += historyInfectedPerDay[i];
    }
    for (int i = 0; i < historyDeadPerDay.size(); i++) {
```

```
            sumKilled += historyDeadPerDay[i];
        }
        for (int i = 0; i < historyCuredPerDay.size(); i++) {
            sumCured += historyCuredPerDay[i];
        }

        GlobalState::infectedAgents = sumInfected;
        GlobalState::deadAgents = sumKilled;
        GlobalState::curedAgents = sumCured;



        AgentList = DEV_Result_AgentList;
        currentDay++;
    }
int maxInfected = GlobalState::infectedAgents;
 int maxDead = GlobalState::deadAgents;
 int maxCured = GlobalState::curedAgents;
 int sumInfected = 0;
 int sumDead = 0;
 int sumCured = 0;

 bool hasFirstInfectedBeenDetected = false;
 bool hasFirstDeadBeenDetected = false;
 bool hasFirstCuredBeenDetected = false;
  bool hasHalfPopulationInfectedBeenDetected = false;
 bool hasHalfPopulationDeadBeenDetected = false;
 bool hasHalfPopulationCuredBeenDetected = false;

 bool hasAllPopulationInfectedBeenDetected = false;
 bool hasAllPopulationDeadBeenDetected = false;
 bool hasAllPopulationCuredBeenDetected = false;

 for(int i = 0; i < GlobalState::Dmax; i++){
   int curDayInfected  = historyInfectedPerDay[i];
   int curDayDead  = historyDeadPerDay[i];
   int curDayCured  = historyCuredPerDay[i];

   sumInfected += curDayInfected;
   sumDead += curDayDead;
   sumCured += curDayCured;
```

```
/* Patients Zero */
if(!hasFirstInfectedBeenDetected){
  if(curDayInfected > 0){
    GlobalState::patientZeroDay = i+1;
    hasFirstInfectedBeenDetected = true;
  }
}
if(!hasFirstDeadBeenDetected){
  if(curDayDead > 0){
    GlobalState::patientZeroDeadDay = i+1;
    hasFirstDeadBeenDetected = true;
  }
}
if(!hasFirstCuredBeenDetected){
  if(curDayCured > 0){
    GlobalState::patientZeroCuredDay = i+1;
    hasFirstCuredBeenDetected = true;
  }
}

/* Half population */
if(!hasHalfPopulationInfectedBeenDetected){
  if(sumInfected > (maxInfected/2) ){
    GlobalState::halfPopulationInfectedDay = i+1;
    hasHalfPopulationInfectedBeenDetected = true;
  }
}
if(!hasHalfPopulationDeadBeenDetected){
  if(sumDead > (maxDead/2) ){
    GlobalState::halfPopulationDeadDay = i+1;
    hasHalfPopulationDeadBeenDetected = true;
  }
}
if(!hasHalfPopulationCuredBeenDetected){
  if(sumCured > (maxCured/2) ){
    GlobalState::halfPopulationCuredDay = i+1;
    hasHalfPopulationCuredBeenDetected = true;
  }
}
```

```
  /* All population */
  if(!hasAllPopulationInfectedBeenDetected){
    if(sumInfected >= maxInfected ){
      GlobalState::fullPopulationInfectedDay = i+1;
      hasAllPopulationInfectedBeenDetected = true;
    }
  }
  if(!hasAllPopulationDeadBeenDetected){
    if(sumDead >= maxDead ){
      GlobalState::fullPopulationDeadDay = i+1;
      hasAllPopulationDeadBeenDetected = true;
    }
  }
  if(!hasAllPopulationCuredBeenDetected){
    if(sumCured >= maxCured ){
      GlobalState::fullPopulationCuredDay = i+1;
      hasAllPopulationCuredBeenDetected = true;
    }
  }

}



   /* END */
   printf("Phase End\n");
   clock_t total_end_CPU = clock();
   float total_elapsedTime_CPU = total_end_CPU - total_start_CPU;
   printf("\tTotal Time CPU: %f ms.\n", total_elapsedTime_CPU);
   printf("===\nTotal Agents: %d\n===\n", GlobalState::N);
   printf("Total Infected Agents: %d\n", GlobalState::infectedAgents);
   printf("Total Dead Agents: %d\n", GlobalState::deadAgents);
   printf("Total Cured Agents: %d\n===\n", GlobalState::curedAgents);
   printf("Day of First Agent Infection: %d\n", GlobalState::patientZeroDay);
   printf("Day of First Agent Dead: %d\n", GlobalState::patientZeroDeadDay);
   printf("Day of First Agent Cured: %d\n===\n", GlobalState::patientZeroCuredDay);
   printf("Day of Half Population Infection: %d\n",
GlobalState::halfPopulationInfectedDay);
   printf("Day of Half Population Dead: %d\n", GlobalState::halfPopulationDeadDay);
```

```
    printf("Day of Half Population Cured: %d\n===\n",
GlobalState::halfPopulationCuredDay);
    printf("Day of Full Population Infection: %d\n",
GlobalState::fullPopulationInfectedDay);
    printf("Day of Full Population Dead: %d\n", GlobalState::fullPopulationDeadDay);
    printf("Day of Full Population Cured: %d\n===\n",
GlobalState::fullPopulationCuredDay);

    printf("Agents infected per Day (History):");
    for (int val : historyInfectedPerDay)
    {
        printf(" %d", val);
    }
    printf("\n");
    printf("Agents killed per Day (History):");
    for (int val : historyDeadPerDay)
    {
        printf(" %d", val);
    }
    printf("\n");
    printf("Agents cured per Day (History):");
    for (int val : historyCuredPerDay)
    {
        printf(" %d", val);
    }
    printf("\n");
    return 0;
}
```

- **CPU**
  - ***classes.cpp***

```
#include <vector>

class Agent
{
public:
  // Position
  int x = 0;
  int y = 0;
```

```
  bool hasBeenAccountedFor = false;

  // No infectado = 0
  // Infectado = 1
  // En Cuarentena = -1
  // Muerto = -2
  int S = 0;
  // Probabilidad de Contagio
  float Pcon = 0;
  // Probabilidad de Contagio Externa
  float Pext = 0;
  //Probabilidad mortalidad
  float Pfat = 0;
  // Probabilidad de movilidad
  float Pmov = 0;
  //Probabilidad de movilidad en pequeñas distancias
  float Psmo = 0;
  //Tiempo de incubación
  int Tinc = 0;
  //Tiempo de recuperación
  int Trec = 14;
  //Constructor
  Agent(int x, int y);
  Agent() = default;
  //Rango de probabilidad de contagio
  static float PconRange[2];
  //Rango de probabilidad externa de contagio
  static float PextRange[2];
  //Rango de probabilidad de mortalidad
  static float PfatRange[2];
  // Rango de probabilidad de movilidad
  static float PmovRange[2];
  // Rango de probabilidad de Movimiento en pequeña distancia
  static float PsmoRange[2];
  //Rango de tiempo de incubación
  static int TincRange[2];
};

class GlobalState
{
```

```
public:
  //Número de agentes
  static int N;
  //Máximo número de días de simulación
  static int Dmax;
  //Máximo número de movimientos por día
  static int Mmax;
  //Radio máximo permitido para movimientos locales
  static int lmax;
  //Distancia límite de contagio
  static int R;
  //Área de simulación, Dimensión 0:y, Dimensión 1:x
  static int mapSize[2];

  /* Estadisticas */
  static int infectedAgents;
  static int curedAgents;
  static int deadAgents;
  static int patientZeroDay;
  static int halfPopulationInfectedDay;
  static int fullPopulationInfectedDay;
  static int patientZeroCuredDay;
  static int halfPopulationCuredDay;
  static int fullPopulationCuredDay;
  static int patientZeroDeadDay;
  static int halfPopulationDeadDay;
  static int fullPopulationDeadDay;
};

class Operations
{
public:
  //First Operation
  static int hasBeenInfected(Agent *agent, std::vector<Agent *> *AgentList);
  //Second Operation
  static int isShortMovement(Agent *agent);
  static int XMovement(Agent *agent);
  static int YMovement(Agent *agent);
  static bool willMove(Agent *agent);
  //Third Operation
```

```
  static int hasBeenExternalInfected(Agent *agent);
  //Fourth Operation
  static int incubationTime(Agent *agent);
  static int hasSymptoms(Agent *agent);
  static int recuperationTime(Agent *agent);
  //Fifth Operation
  static int isInRecuperation(Agent *agent);
  static int isDead(Agent *agent);
  //Helpers
  static float GenerateRandomFloatBetween(float a, float b);
  static int GenerateRandomIntegerBetween(int a, int b);
  static int Clamp(int value, int low, int high);
};
```

- ***main.cpp***

```
#include <stdio.h>
#include <stdlib.h>
#include <iostream>
#include <cstring>
#include <math.h>
#include <time.h>
#include <random>
#include <chrono>
#include <vector>
#include <string>
#include <sstream>

#include "classes.cpp"

using namespace std;

class Agent;
class Operations;
class GlobalState;

typedef std::chrono::high_resolution_clock myclock;
myclock::time_point beginning = myclock::now();
myclock::duration d = myclock::now() - beginning;
unsigned seed2 = d.count();
std::mt19937 rng(seed2);
std::default_random_engine generator;
```

```
float GenerateRandomFloatBetween(float a, float b)
{
    float res;
    std::uniform_real_distribution<double> distribution(a, b);
    res = distribution(generator);
    return res;
}

int GenerateRandomIntegerBetween(int a, int b)
{
    std::uniform_int_distribution<int> gen(a, b);
    return gen(rng);
}

/*Operations Declarations*/
//First Operation
int hasBeenInfected(Agent *agent, vector<Agent *> *AgentList)
{
    if (agent->S != 0)
    {
        return agent->S;
    }
    int res = 0;
    // Part First Part
    float prob = GenerateRandomFloatBetween(0, 1);
    bool fp = prob <= agent->Pcon ? 1 : 0;
    bool a = 0;

    // Part: Calculate Alpha
    bool sp = 0;
    bool tp = 0;
    int validateFn = 0;
    int sum = 0;

    for (Agent *agent_test : *AgentList)
    {
        if (agent->x == agent_test->x && agent->y == agent_test->y)
        {
            continue;
```

```
        }

        float distance = (float)sqrt(pow(agent_test->x - agent->x, 2) + pow(agent_test->y
- agent->y, 2));
        sp = distance <= GlobalState::R;
        // Part: Calculate Beta
        tp = agent_test->S > 0 ? 1 : 0;
        sum += sp * tp;
    }
    a = sum >= 1 ? 1 : 0;

    return (fp * a) ? 1 : agent->S;
}

//Second Operation
int isShortMovement(Agent *agent)
{
    float prob = GenerateRandomFloatBetween(0, 1);
    return (prob <= agent->Psmo) ? 1 : 0;
}

int XMovement(Agent *agent)
{
    int s = isShortMovement(agent);
    int p = GlobalState::mapSize[1];
    float prob = GenerateRandomFloatBetween(0, 1);
    float longDistance = p * prob * (1 - s);
    prob = GenerateRandomFloatBetween(0, 1);
    float shortDistance = (agent->x + (2 * prob - 1) * GlobalState::lmax) * s;
    return shortDistance + longDistance;
}

int YMovement(Agent *agent)
{
    int s = isShortMovement(agent);
    int q = GlobalState::mapSize[0];
    float prob = GenerateRandomFloatBetween(0, 1);
    float longDistance = q * prob * (1 - s);
    prob = GenerateRandomFloatBetween(0, 1);
    float shortDistance = (agent->y + (2 * prob - 1) * GlobalState::lmax) * s;
```

```
    return shortDistance + longDistance;
}

bool willMove(Agent *agent)
{
    float prob = GenerateRandomFloatBetween(0, 1);
    return (prob <= agent->Pmov) ? 1 : 0;
}

//Third Operation
int hasBeenExternalInfected(Agent *agent)
{
    float prob = GenerateRandomFloatBetween(0, 1);
    //Part: Calculate First Part
    bool fp = prob <= agent->Pext ? 1 : 0;
    //Part: Calcula Epsilon
    bool sp = agent->S != 0 ? 0 : 1;
    //fp * sp == if fp or sp are 0 the condition is false
    bool finalB = (fp * sp) > 0;

    return (finalB) ? 1 : agent->S;
}

//Fourth Operation
int incubationTime(Agent *agent)
{
    int Tinc = agent->Tinc;
    if (agent->S > 0)
    {
        Tinc--;
    }
    return Tinc;
}

int hasSymptoms(Agent *agent)
{
    int S = -1;
    if (agent->Tinc > 0)
    {
        S = agent->S;
```

```
    }
    return S;
}

int recuperationTime(Agent *agent)
{
    int Trec = agent->Trec;
    if (agent->S == -1)
    {
        Trec--;
    }
    return Trec;
}

//Fifth Operation
int isInRecuperation(Agent *agent)
{
    return (agent->S < 0) ? 1 : 0;
}

int isDead(Agent *agent)
{
    int o = isInRecuperation(agent);
    float prob = GenerateRandomFloatBetween(0, 1);
    int Pfat = (prob <= agent->Pfat) ? 1 : 0;
    return (Pfat * o > 0) ? -2 : agent->S;
}

int Clamp(int value, int low, int high)
{
    return value > high ? high : value < low ? low : value;
}

/*GlobalState Declarations*/
int GlobalState::mapSize[2] = {500, 500};
int GlobalState::Dmax = 60;
int GlobalState::Mmax = 10;
int GlobalState::lmax = 5;
int GlobalState::N = 10240;
int GlobalState::infectedAgents = 0;
```

```
int GlobalState::R = 1;
/* Statistics*/
int GlobalState::curedAgents = 0;
int GlobalState::deadAgents = 0;
int GlobalState::patientZeroDay = 0;
int GlobalState::halfPopulationInfectedDay = 0;
int GlobalState::fullPopulationInfectedDay = 0;
int GlobalState::patientZeroCuredDay = 0;
int GlobalState::halfPopulationCuredDay = 0;
int GlobalState::fullPopulationCuredDay = 0;
int GlobalState::patientZeroDeadDay = 0;
int GlobalState::halfPopulationDeadDay = 0;
int GlobalState::fullPopulationDeadDay = 0;

float Agent::PconRange[2] = {0.02, 0.03};
float Agent::PextRange[2] = {0.02, 0.03};
float Agent::PfatRange[2] = {0.007, 0.07};
float Agent::PmovRange[2] = {0.3, 0.5};
float Agent::PsmoRange[2] = {0.7, 0.9};
int Agent::TincRange[2] = {5, 6};

/*Agent Constructor*/
Agent::Agent(int x, int y)
{
    // Position
    Agent::x = x;
    Agent::y = y;
    // Assing the Infection Probability
    Agent::Pcon = GenerateRandomFloatBetween(Agent::PconRange[0], Agent::PconRange[1]);
    // Assign the External Infection Probability
    Agent::Pext = GenerateRandomFloatBetween(Agent::PextRange[0], Agent::PextRange[1]);
    //Assign the Mortality Probability
    Agent::Pfat = GenerateRandomFloatBetween(Agent::PfatRange[0], Agent::PfatRange[1]);
    // Assign the Movement Probability
    Agent::Pmov = GenerateRandomFloatBetween(Agent::PmovRange[0], Agent::PmovRange[1]);
    // Assign the Small Movement Probability
    Agent::Psmo = GenerateRandomFloatBetween(Agent::PsmoRange[0], Agent::PsmoRange[1]);
    // Assign the Incubation Time
    Agent::Tinc = GenerateRandomFloatBetween(Agent::TincRange[0], Agent::TincRange[1]);
};
```

```
/**/
int main()
{
    clock_t total_start_CPU = clock();

    /* 1. Map Generation and Setup */
    int size0 = 500;
    int size1 = 500;
    vector<int> historyInfectedPerDay;
    vector<int> historyCuredPerDay;
    vector<int> historyDeadPerDay;

    vector<Agent *> AgentList;
    Agent **AgentMap = new Agent *[500 * 500];

    for (int i = 0; i < size0; i++)
    {
        for (int j = 0; j < size1; j++)
        {
            AgentMap[i * 500 + j] = NULL;
        }
    }

    /* 2. Generate Agents on Map */
    int leftAgents = GlobalState::N;
    while (leftAgents > 0)
    {
        int x = GenerateRandomIntegerBetween(0, 499);
        int y = GenerateRandomIntegerBetween(0, 499);

        if (AgentMap[y * 500 + x] != NULL)
        {
            continue;
        }

        Agent *agent = new Agent(x, y);
        AgentMap[y * 500 + x] = agent;
        AgentList.push_back(agent);
        leftAgents--;
```

```
    }

    /* Operation Phase*/
    int currentDay = 0;
    //Mientras que el día actual de simulación sea menor que dMax
    while (currentDay < GlobalState::Dmax)
    {
        int infectedToday = 0;
        int killedToday = 0;
        int curedToday = 0;
        int currentMovement = 0;
        //Mientras que el movimiento actual sea menor que mMax
        while (currentMovement < GlobalState::Mmax)
        {
            //Para todos los agentes
            for (int i = 0; i < AgentList.size(); i++)
            {
                Agent *agent = AgentList[i];
                int previousState = agent->S;

                //Aplicar Regla 1
                vector<Agent *> Neighbours;
                if (agent->y > 0 && AgentMap[(agent->y - 1) * 500 + (agent->x)] != NULL)
                {
                    Neighbours.push_back(AgentMap[(agent->y - 1) * 500 + (agent->x)]);
                }
                if (agent->x > 0 && AgentMap[(agent->y) * 500 + (agent->x - 1)] != NULL)
                {
                    Neighbours.push_back(AgentMap[(agent->y) * 500 + (agent->x - 1)]);
                }
                if (agent->x < size0 - 1 && AgentMap[(agent->y) * 500 + (agent->x + 1)] !=
NULL)
                {
                    Neighbours.push_back(AgentMap[(agent->y) * 500 + (agent->x + 1)]);
                }
                if (agent->y < size1 - 1 && AgentMap[(agent->y + 1) * 500 + (agent->x)] !=
NULL)
                {
                    Neighbours.push_back(AgentMap[(agent->y + 1) * 500 + (agent->x)]);
                }
```

```
agent->S = hasBeenInfected(agent, &Neighbours);

//Aplicar Regla 2
if (willMove(agent))
{
    int x;
    int y;

    do
    {
        x = XMovement(agent);
        y = YMovement(agent);

        x = Clamp(x, 0, size0 - 1);
        y = Clamp(y, 0, size1 - 1);
    } while (AgentMap[y * 500 + x] != NULL);

    //Se elimina de la vieja posición
    AgentMap[agent->y * 500 + agent->x] = NULL;

    //Se añade a la nueva posición
    agent->x = x;
    agent->y = y;
    AgentMap[y * 500 + x] = agent;
}

if (agent->S != previousState)
{
    switch (agent->S)
    {
    case 1: //Infected
        if (GlobalState::infectedAgents == 0)
        {
            GlobalState::patientZeroDay = currentDay;
        }
        GlobalState::infectedAgents++;
        infectedToday++;
        break;
    }
}
```

```
            }
            currentMovement++;
        }

        for (int i = 0; i < AgentList.size(); i++)
        {
            Agent *agent = AgentList[i];
            int previousState = agent->S;

            //Aplicar Regla 3
            agent->S = hasBeenExternalInfected(agent);

            //Aplicar Regla 4
            agent->Tinc = incubationTime(agent);
            agent->S = hasSymptoms(agent);
            agent->Trec = recuperationTime(agent);
            if (agent->Trec <= 0)
            {
                agent->Pcon = -1;
                agent->Pext = -1;
                agent->Pfat = -1;
                agent->S = 0;

                if (!agent->hasBeenAccountedFor)
                {
                    if (GlobalState::curedAgents == 0)
                    {
                        GlobalState::patientZeroCuredDay = currentDay;
                    }
                    GlobalState::curedAgents++;
                    curedToday++;
                    agent->hasBeenAccountedFor = true;
                }
            }
            //Aplicar Regla 5
            agent->S = isDead(agent);

            if (agent->S != previousState)
            {
                switch (agent->S)
```

```
                {
                case 1: //Infected
                    if (GlobalState::infectedAgents == 0)
                    {
                        GlobalState::patientZeroDay = currentDay;
                    }
                    GlobalState::infectedAgents++;
                    infectedToday++;
                    break;
                case -2: // Dead
                    agent->Pcon = -1;
                    agent->Pext = -1;
                    agent->Pfat = -1;
                    agent->S = 0;
                    agent->hasBeenAccountedFor = true; //TEST

                    if (GlobalState::deadAgents == 0)
                    {
                        GlobalState::patientZeroDeadDay = currentDay;
                    }
                    GlobalState::deadAgents++;
                    killedToday++;
                    break;
                }
            }
        }

        historyCuredPerDay.push_back(curedToday);
        historyDeadPerDay.push_back(killedToday);
        historyInfectedPerDay.push_back(infectedToday);

        curedToday = 0;
        killedToday = 0;
        infectedToday = 0;
        currentDay++;
    }
int maxInfected = GlobalState::infectedAgents;
 int maxDead = GlobalState::deadAgents;
 int maxCured = GlobalState::curedAgents;
 int sumInfected = 0;
```

```
int sumDead = 0;
int sumCured = 0;

bool hasFirstInfectedBeenDetected = false;
bool hasFirstDeadBeenDetected = false;
bool hasFirstCuredBeenDetected = false;
 bool hasHalfPopulationInfectedBeenDetected = false;
bool hasHalfPopulationDeadBeenDetected = false;
bool hasHalfPopulationCuredBeenDetected = false;

bool hasAllPopulationInfectedBeenDetected = false;
bool hasAllPopulationDeadBeenDetected = false;
bool hasAllPopulationCuredBeenDetected = false;

for(int i = 0; i < GlobalState::Dmax; i++){
  int curDayInfected  = historyInfectedPerDay[i];
  int curDayDead  = historyDeadPerDay[i];
  int curDayCured  = historyCuredPerDay[i];

  sumInfected += curDayInfected;
  sumDead += curDayDead;
  sumCured += curDayCured;

  /* Patients Zero */
  if(!hasFirstInfectedBeenDetected){
    if(curDayInfected > 0){
      GlobalState::patientZeroDay = i+1;
      hasFirstInfectedBeenDetected = true;
    }
  }
  if(!hasFirstDeadBeenDetected){
    if(curDayDead > 0){
      GlobalState::patientZeroDeadDay = i+1;
      hasFirstDeadBeenDetected = true;
    }
  }
  if(!hasFirstCuredBeenDetected){
    if(curDayCured > 0){
      GlobalState::patientZeroCuredDay = i+1;
      hasFirstCuredBeenDetected = true;
```

```
    }
  }

  /* Half population */
  if(!hasHalfPopulationInfectedBeenDetected){
    if(sumInfected > (maxInfected/2) ){
      GlobalState::halfPopulationInfectedDay = i+1;
      hasHalfPopulationInfectedBeenDetected = true;
    }
  }
  if(!hasHalfPopulationDeadBeenDetected){
    if(sumDead > (maxDead/2) ){
      GlobalState::halfPopulationDeadDay = i+1;
      hasHalfPopulationDeadBeenDetected = true;
    }
  }
  if(!hasHalfPopulationCuredBeenDetected){
    if(sumCured > (maxCured/2) ){
      GlobalState::halfPopulationCuredDay = i+1;
      hasHalfPopulationCuredBeenDetected = true;
    }
  }

  /* All population */
  if(!hasAllPopulationInfectedBeenDetected){
    if(sumInfected >= maxInfected ){
      GlobalState::fullPopulationInfectedDay = i+1;
      hasAllPopulationInfectedBeenDetected = true;
    }
  }
  if(!hasAllPopulationDeadBeenDetected){
    if(sumDead >= maxDead ){
      GlobalState::fullPopulationDeadDay = i+1;
      hasAllPopulationDeadBeenDetected = true;
    }
  }
  if(!hasAllPopulationCuredBeenDetected){
    if(sumCured >= maxCured ){
      GlobalState::fullPopulationCuredDay = i+1;
      hasAllPopulationCuredBeenDetected = true;
```

```
    }
  }

}



    /* END */
    printf("Phase End\n");
    clock_t total_end_CPU = clock();
    float total_elapsedTime_CPU = total_end_CPU - total_start_CPU;
    printf("\tTotal Time CPU: %f ms.\n", total_elapsedTime_CPU);
    printf("===\nTotal Agents: %d\n===\n", GlobalState::N);
    printf("Total Infected Agents: %d\n", GlobalState::infectedAgents);
    printf("Total Dead Agents: %d\n", GlobalState::deadAgents);
    printf("Total Cured Agents: %d\n===\n", GlobalState::curedAgents);
    printf("Day of First Agent Infection: %d\n", GlobalState::patientZeroDay);
    printf("Day of First Agent Dead: %d\n", GlobalState::patientZeroDeadDay);
    printf("Day of First Agent Cured: %d\n===\n", GlobalState::patientZeroCuredDay);
    printf("Day of Half Population Infection: %d\n",
GlobalState::halfPopulationInfectedDay);
    printf("Day of Half Population Dead: %d\n", GlobalState::halfPopulationDeadDay);
    printf("Day of Half Population Cured: %d\n===\n",
GlobalState::halfPopulationCuredDay);
    printf("Day of Full Population Infection: %d\n",
GlobalState::fullPopulationInfectedDay);
    printf("Day of Full Population Dead: %d\n", GlobalState::fullPopulationDeadDay);
    printf("Day of Full Population Cured: %d\n===\n",
GlobalState::fullPopulationCuredDay);

    printf("Agents infected per Day (History):");
    for (int val : historyInfectedPerDay)
    {
        printf(" %d", val);
    }
    printf("\n");
    printf("Agents killed per Day (History):");
    for (int val : historyDeadPerDay)
    {
        printf(" %d", val);
```

```
    }
    printf("\n");
    printf("Agents cured per Day (History):");
    for (int val : historyCuredPerDay)
    {
        printf(" %d", val);
    }
    printf("\n");
    return 0;
}
```

RESULTADOS

Agrega las imágenes obtenidas en los espacios indicados.

```
===
Total Agents: 10240
===
Total Infected Agents: 8264
Total Dead Agents: 2987
Total Cured Agents: 4383
===
Day of First Agent Infection: 1
Day of First Agent Dead: 5
Day of First Agent Cured: 18
===
Day of Half Population Infection: 19
Day of Half Population Dead: 26
Day of Half Population Cured: 33
===
Day of Full Population Infection: 60
Day of Full Population Dead: 60
Day of Full Population Cured: 60
===
Agents infected per Day (History): 274 243 272 281 270 246 261 237 262 199 192 199 21
5 226 188 180 187 174 192 176 164 151 135 154 126 138 127 139 124 123 116 127 128 96
112 112 79 93 86 78 88 99 84 77 80 67 77 76 63 73 70 62 62 75 58 61 50 55 45 60
Agents killed per Day (History): 0 0 0 0 13 21 37 37 41 54 65 71 73 80 83 84 103 87 8
1 67 98 92 89 71 80 79 59 65 67 59 72 58 53 49 57 49 47 58 45 45 47 38 31 35 37 44 39
 35 42 30 31 46 33 26 41 28 32 22 35 26
Agents cured per Day (History): 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 171 147 182 167 164
 140 148 141 161 116 126 133 118 150 120 109 120 110 122 105 97 89 86 96 69 89 76
69 81 76 74 81 63 69 70 41 62 53 38 63 53 47
```

Imagen de la consola con los resultados

```
Phase End
    Total Time CPU: 8706208.000000 ms.
===
```

```
Phase End
        Total Time GPU: 2917.000000 ms.
```

Imagen de la consola en la que se muestre el tiempo de ejecución en el CPU	Imagen de la consola en la que se muestre el tiempo de ejecución en el GPU

CONCLUSIONES

Escribe tus observaciones y conclusiones.
Este proyecto significó un reto para nosotros primeramente porque el conocimiento sobre C++ que tenemos nos estuvo limitando a lo largo de la implementación. En segunda nos fuimos encontrando con diferentes errores en CUDA sobre todo sobre la manera de mandar al device y al kernel la información de manera correcta, copiarla adecuadamente y modificarla de manera apropiada. Al final estuvimos comparando implementaciones y dependiendo del número de kernels que utilizabamos, la combinación entre CPU y GPU nos daba diferentes tiempos entonces fue cuando nos dimos cuenta que sería más rápido tener una combinación de CPU y GPU entonces las reglas 1 y 2 las se hicieron en la primera y las reglas 3, 4 y 5 en la segunda. El reto nos mantuvo activos y nos permitió aplicar e incrementar nuestros conocimientos en C++ y sobre todo en CUDA. Los resultados sorprenden y nos permite darnos una idea cómo se realizan los estudios de comportamiento en muchas situaciones así como la actual pandemia. CUDA es pues una gran herramienta para poder reducir los tiempos de proceso..