Ingeniería de Computadores


Miembros del grupo:

- Elvi Mihai Sabau Sabau
- Ander Dorado Bolé
- Daniel Asensi Roch

# Práctica 2

20 de octubre de 2021

# Índice

# 1. Tarea 1: Búsqueda del problema

En esta tarea dedicaremos un tiempo para buscar un problema paralelizable, que se pueda aplicar a la vida real. En nuestro caso, sera el juego de la vida, que se aplica a la simulación de células y a la automatización celular

## 1.1. Objetivos

Game of Life, es un autómata celular ideado por el matemático británico John Horton Conway en 1970. Es un juego de cero jugadores, lo que significa que su evolución está determinada por su estado inicial y no requiere más información. Se interactúa con el Juego de la Vida creando una configuración inicial y observando cómo evoluciona. Es Turing completo y puede simular un constructor universal o cualquier otra máquina de Turing.

## 1.2. Implementación

El programa recibirá como parámetro el número de generaciones a procesar, y empezará a realizar generaciones de células a partir de las originales y del dominio en el que se procesa.

El juego finalizará cuando llegue al límite de generaciones.

### 1.2.1. Inicio

El programa carga una matriz que usará de tablero, con valores a 0 o 1, siendo 0 las células muertas y 1 las células vivas. Esta matriz será el estado inicial de nuestro algoritmo.

Una vez cargado el estado inicial, el algoritmo modifican los valores de cada matriz según el valor de la suma de los valores circundantes:

- Si la celda actual está muerta y el resultado es 3, la celda actual nace, pasando su estado de muerta (0) a viva (1).
- Si la celda actual está viva y el resultado es 2 o 3, esta sobrevive (mantiene su estado actual)
- En cualquier otro caso, la celda actual muere (pasa a 0)

### 1.2.2. Secuencialmente

1. Se recorre secuencialmente la matriz de estados iniciales.
2. Para cada estado/valor se suman los estados/valores circundantes a este.
3. Según el estado de la celda y la suma de los valores se modificará o se mantendrá el estado/valor.
4. Una vez finalizada la comprobación vuelve a repetir este proceso por cada ciclo.

### 1.2.3. Paralelización

El programa divide paralelamente **(datos)** cada celda/estado de un ciclo. Por cada una de estas celdas se dividen las celdas circundantes dos a dos y se realiza la suma de cada una de manera paralela **(funcional)**. Una vez obtenidos los resultados se vuelve a repetir el proceso de la suma hasta obtener un único valor. Se repite este proceso por cada ciclo.

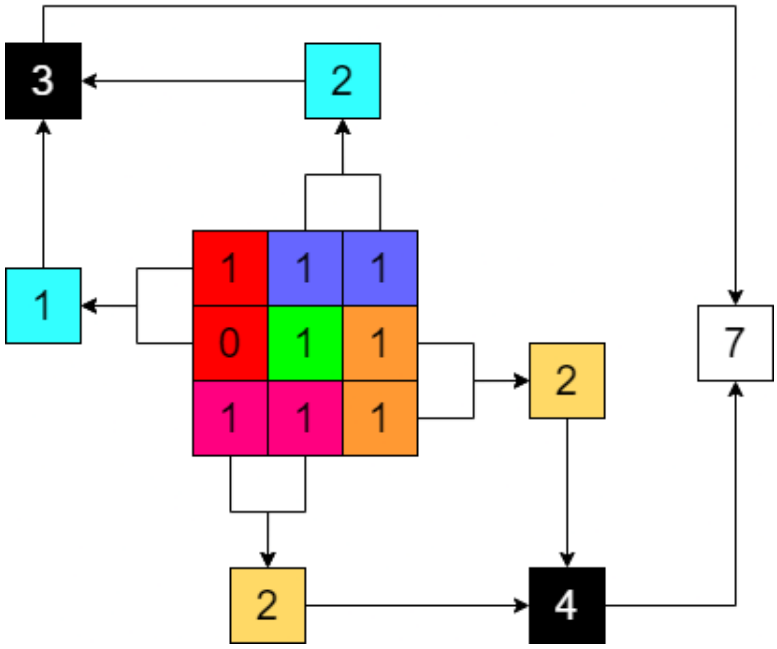



Figura 1: Representación de la suma paralela por cada celda.

#### 1.2.4. Final

Por cada generación, se imprime la matriz con el estado de cada celda, una vez alcanzado el número máximo de iteraciones, el programa finaliza.

# 2. Tarea 2: Estudio del problema

## 2.1. Grafo control de flujo

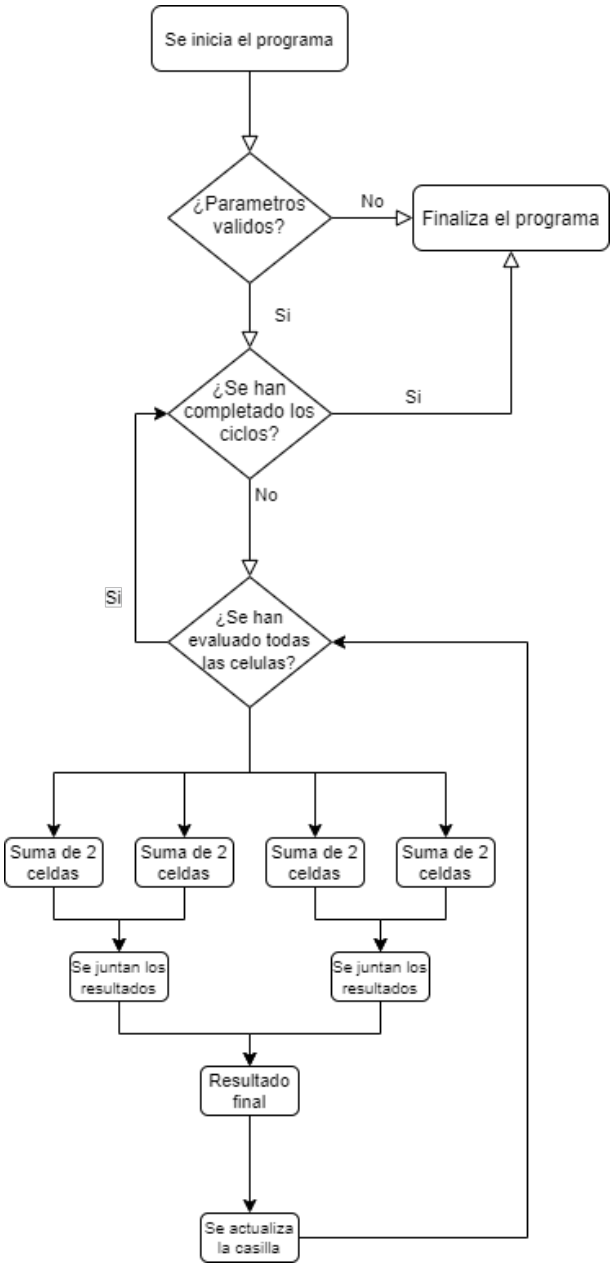



Figura 2: Gráfico control de flujo

## 2.2. Estructura del programa

La estructura del código se separa en las siguientes partes.

Las estructuras de datos y variables:

- Un vector de filas y otro de columnas, representado cada x,y una casilla que alberga una casilla viva o

muerta.

- El tablero, un array de números, que representan el tablero, siendo 1 las células vivas y 0 las células muertas, dando el estado inicial a la primera generación del juego.
- Una variable de enteros que representa el columnas, y otra el numero de filas.
- Y por último, otra variable de enteros que especifica el número total de generaciones a procesar, y otra que define el tiempo total de espera entre generaciones para poder visualizar en vivo la evolución.

Funciones:

- getNeighbor: Devuelve las coordenadas de las casillas vecinas a una casilla pasada por parametro.
- sumarVecinos: Cuenta las células vecinas. Usa de la función getNeighbor.
- mostrarCelda: Imprime una celda por pantalla.
- mostrarFila: Imprime una fila por pantalla.
- mostrarCeldas: Imprime varias celdas por pantalla. Usa de la función mostrarCelda.
- tick: Establece un estado para una célula, dado el resultado de los cálculos de sus células vecinas
- updateCeldas: Procesa cada una de las celdas y establece su nuevo estado. Usa de la función tick.
- startGame: Código principal que arranca el programa, realizando el procesamiento del tablero un número total de n generaciones.

## 2.3. Variables que afectan la carga de nuestro problema

Las variables que afectan a la carga de nuestro programa son en general:

- Las dimensiones de la matriz.
- La cantidad de células vivas.
- El número de generaciones.
- El número de cálculos a realizar.

# 3. Tarea 3: Sintonización del compilador y análisis del rendimiento del programa

Debemos tener en cuenta que el programa está escrito en C++ y ha sido compilado en una máquina con 8GB de RAM, 8 núcleos y en Ubuntu 20.04, el valor obtenido depende de las características de esta máquina.

Probamos algunos parámetros del compilador, cuyo propósito es optimizar el código compilado (g ++ -help = estos parámetros serán mostrados por el optimizador) o habilitando ciertas herramientas (g ++ -help = target).

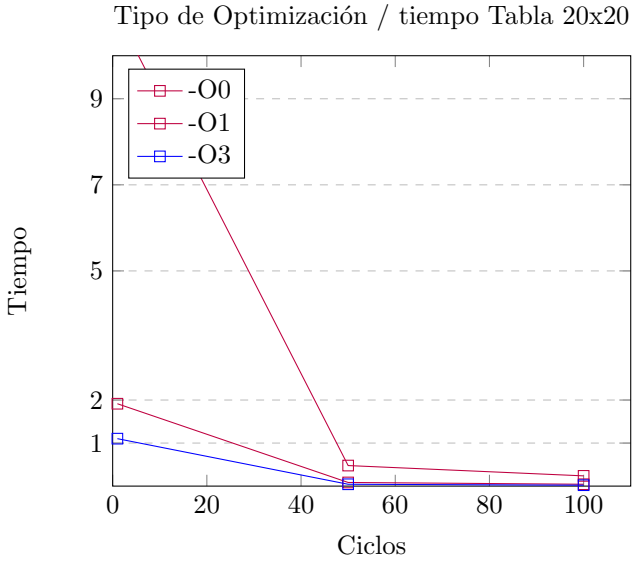
En nuestro programa, nos centramos en la optimización utilizando parámetros de tipo -O, especialmente -O0, -O1, -O3.

## 3.1. Lista de parámetros y opciones del compilador

- **-O0 estándar**, reduce el tiempo de compilación y depura.
- **-O1** intenta reducir el tamaño del código y el tiempo de ejecución.
- **-O2** mayor optimización, casi todas las optimizaciones admitidas se han llevado a cabo. El compilador elimina el intercambio de space-speed.
- **-O3** Incluye las optimizaciones de -O2 incluyendo las anteriores.
- **-Os** Incluye las optimizaciones de -O2 excepto las que aumenten el tamaño del código.
- **-Ofast** Incluye las optimizaciones de -O3 ignorando el estándar.
- **-Og** Optimización orientada a la experiencia de debugging, activando flags de recolección de información que en -O0 no están habilitados por defecto.

## 3.2. Análisis de optimización del uso de los parámetros

A través de los parámetros utilizados, notamos que para una variable con un valor relativamente pequeño (tamaño de matriz), el programa gana al ser compilado con -O1 comparado -O0.




Podemos asumir el código compilado con los parámetros -O1 y -O3 optimizará el código y su velocidad de ejecución disminuirá relativamente a medida que el problema crezca, aunque como podemos ver en la grafica, no hay una gran diferencia entre -O1 y -O3.

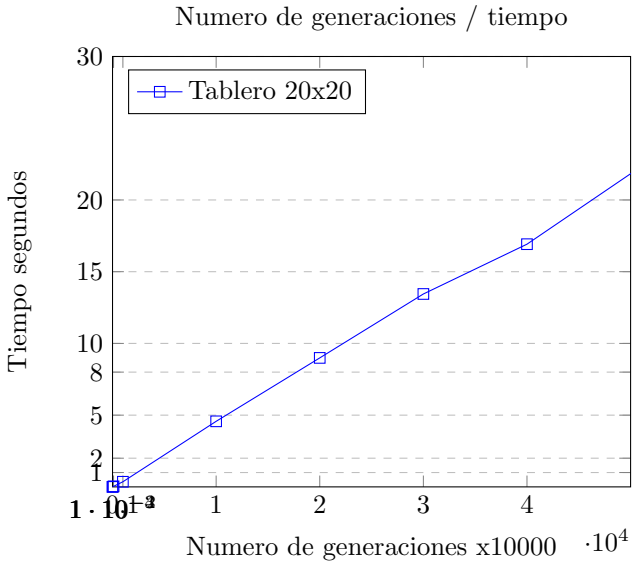
### 3.3. Como cambia el speed-up dependiendo de las variables de nuestro problema

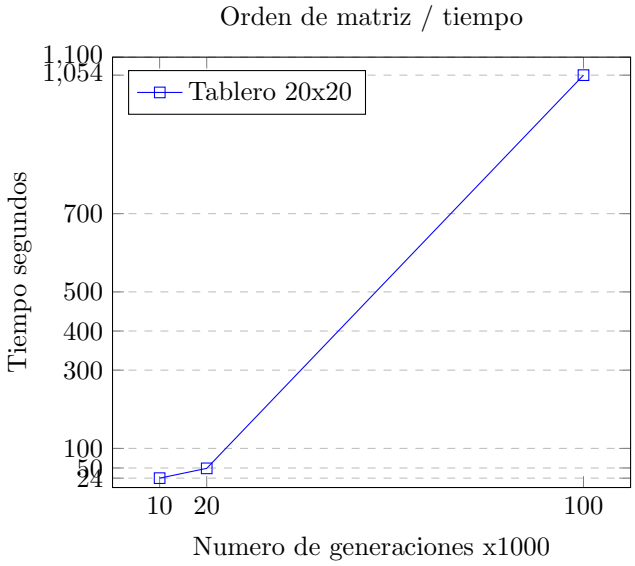
El speed-up vendrá dada en nuestra aplicación por 3 inputs definidos en el propio código del programa.

- Tamaño de ecosistema: El tamaño del ecosistema vendrá dado por el orden la matriz,en ella podremos ver la cantidad de celdas disponibles para nuestras células, será en esta variable donde podamos ver un mayor speed-up en la paralelización de nuestro programa, ya que al procesar las celdas dos a dos en cada uno de los hilos del procesador.
- Ecosistema: En primer lugar dependeremos de la variable que identificaremos como el tablero de juego, en ella definiremos la generación inicial de nuestro ecosistema, donde colocaremos la celdas que deseemos que se inicialicen como vivas en valor 1 y las deseadas como muertas a valor 0, este tablero irá actualizándose dependiendo de los valores de las celdas vivas que posea alrededor cada uno de los componentes de la matriz actualizándose en cada una de las generaciones.
- Generaciones: esta variable será la encargada de contar los ciclos de evolución de nuestro programa, con ella estableceremos el limite de actualizaciones que realizará nuestro ecosistema, a mayor número de generaciones mayor tiempo de ejecución.

### 3.4. Análisis del rendimiento de las variaciones del speed-up

En el análisis del rendimiento podemos observar que la complejidad de nuestro programa es claramente de O(n), ya que a mayor cantidad de elementos para recorrer en la matriz mayor es el tiempo de ejecución de nuestro programa.


Numero de generaciones / tiempo
Tablero 20x20
Tiempo segundos
30
20
15
10
8
5
2
1
1 · 10^-3
1
2
3
4
Numero de generaciones x10000
·10^4


Orden de matriz / tiempo
Tablero 20x20
Tiempo segundos
1,100
1,054
700
500
400
300
100
50
24
10
20
100
Numero de generaciones x1000