INGENIERIA DE LOS COMPUTADORES

PRÁCTICA 2

Estudio y propuesta de la paralelización de una aplicación

JUAN ANTONIO BONILLO
MATILDE ORTÍN INSIGNARES
HELENA LOSA MARTÍNEZ
ALEJANDRO ROS ARLANZÓN
ADAN PLAZA SERRANO

Ingeniería Informática 3º

ÍNDICE

I.INTRODUCCIÓN

En este trabajo miraremos que la paralelización tanto funcional como de datos son unas de las principales formas de acelerar la ejecución de los programas. Por ese motivo, a lo largo de la práctica se paralelizará el Conway's game of life, que es una especie de simulador que representa la vida de las células, en el que añadimos los cálculos del promedio y distribución de vecinos y la estabilidad del tablero.

Empezaremos identificando a través de un grafo de dependencias, las mismas existentes en las funciones, que identificaran si se pueden variar el orden de los elementos para que no varíe la solución. Además también analizaremos los accesos de lectura/escritura de las variables, por otro lado miraremos el estudio de variación de carga del problema. Para finalizar el análisis del programa, estudiaremos cómo varían los tiempos de ejecución del programa con distintos tamaños de tableros, que es cómo podemos hacer que se ralentice el programa. Además de modificar la compilación del archivo para poner distintos targets para ver si se optimiza el tiempo.

Por último se hará la realización de un problema propuesto en el enunciado de la Práctica 2, que es un precalentamiento para las siguientes prácticas, donde calcularemos los tiempos de llenado de un depósito con dos grifos.

II. PROBLEMA ELEGIDO

Este apartado tiene como intención informar de cuáles han sido los motivos para escoger el problema del juego Conway's game of life. Para esto hay que tener en cuenta que para poder realizar el trabajo, se nos pedía que el problema escogido sea uno que “tarde” a nivel de ejecución, y que a la vez este presentará dependencias funcionales y de datos.

En este caso el problema escogido Conway’s game of live presenta dependencia funcional, esto lo podemos ver cuando el estado siguiente de cada celda está determinado por el estado actual y el de sus vecinos, por otro lado para poder cumplir con la parte de dependencia de datos se han añadido tres métricas que dependen una de la otra, la primera función vendría a ser promedi_vecinos que calcula el número promedio de vecinos vivos que tiene las casillas vivas, la segunda función es la distribucion_vecinos, que usa los datos obtenidos en la primera función y devuelve el índice de dispersión, la última función vendría a ser la estabilidad que usa los datos obtenidos en la segunda función para poder calcular cómo de estable es la población de una generación a otra.

Ahora vamos a explicar porque el Conway’s game of life es una buena elección para paralelizar. Podemos ver que su tiempo de ejecución va a ser un tiempo elevado, en tableros con dimensiones grandes y números elevados en iteraciones. Por otro lado tenemos el cálculo de cada celda que se encuentra dentro de una misma generación, esto permite que se pueda realizar de forma independiente.

III. ESTUDIO DEL PROBLEMA CONWAY'S GAME OF LIFE

III.I GRAFO DE DEPENDENCIAS ENTRE TAREAS

En este punto se procederá a mostrar y a explicar el grafo obtenido de dependencias entre tareas, explicando en qué consiste cada tarea. Y también hablaremos de los "trozos" que su ejecución puede ser paralela, es decir si se permuta el orden de ejecución el resultado obtenido no varía.

$$T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow T_4 \rightarrow T_5 \rightarrow T_6$$

T_1: Inicialización del programa y del tablero. (Esto lo encontramos en los métodos initializeGrid y initializedGridFromFile).

T_2: Cálculo de número de vecinos vivos que hay por cada celda. (Esto se encuentra en el método nextLiveCells).

T_3: Actualizar el estado de cada celda para prepararla para la siguiente generación.

T_4: Contar las celdas vivas. (Esto estaría en el método countLiveCells).

T_5: Cálculo de las métricas promedio de vecinos, densidad y estabilidad. (Esto estaría en las funciones promedi_vecinos, distribucion_vecinos y estabilidad).

T_6: Repetición del proceso para cada generación (Esto se encuentra en el método start)

Ahora vamos a analizar en qué parte del grafo de dependencias se pueden encontrar "trozos" que se pueden paralelizar, esto lo podemos ver en dos parte la primera vendría a ser en el "trozo" T_2, que consiste en calcular el número de vecinos vivos que se encuentra en cada celda, que se podría realizar en paralelo porque para su cálculo no se necesita la actualización de las células. El segundo "trozo" paralelizable vendría a ser T_3 ya que la actualización del estado de la celda depende de la generación anterior, no de las celdas que hayan sido actualizadas.

Por tanto podemos ver que los "trozos" paralelizables vendrían a ser T_2 y T_3, donde el orden de ejecución de las operaciones en cada celda no estaría alterando el resultado final.

III.II ESTUDIO DE LAS VARIABLES

En este apartado vamos a estudiar cómo es el acceso a las variables de lectura\escritura más importantes, es decir a las que más se acceden y también estudiaremos las de accesos puntuales. Estudiaremos cómo son estas variables, sí se puede estructurar el flujo del programa en forma estructurada y por último si podremos llegar a prever algún problema en la caché.

Ahora procederemos a analizar las variables calientes, las que más se acceden, en este caso se han encontrado dos la primera que es grid es de tipo vector<vector<CellStates>>, que tiene acceso tanto de escritura como de lectura, esta representa la matriz actual del juego, se está constantemente accediendo a ella para poder leer el estado actual del tablero y escribir el siguiente. La segunda variable que tiene un acceso constante es la de neighbors esta es de tipo vector<vector<int>>, esta también tiene acceso de escritura y de lectura, se dedica a guardar el número de vecinos vivos que hay en una celda, esta variable se actualiza cada vez que se actualiza una celda. Una vez explicado cuáles son las variables calientes, vamos a explicar las que se acceden puntualmente, en este caso tenemos la variable totalCells, ésta sería de tipo int su acceso es de lectura, su función principal se encuentra en las métricas para poder normalizar los datos obtenidos. Por último, tendríamos la variable DensityVal, esta es de tipo double su acceso es de escritura, se utiliza en la función start y se utiliza para representar la métrica de estabilidad entre generaciones.

Como ya se han explicado los diferentes tipos de variables que encontramos en nuestro problema, ahora vamos a estudiar, si hubiera una forma más apropiada de estructurar el código. En este caso podemos decir que sí se puede reestructurar el código de una forma más apropiada, por ejemplo paralelizar los bucles que calculan nextLiveCells y nextGen, ya que estos resultados son independientes entre sí.

Para finalizar, se procederá a explicar sobre los problemas en la caché y cómo los podemos prever. En este caso encontramos que si el tamaño de la cuadrícula del problema es muy grande podría llegar a exceder la capacidad de la caché, por otro lado también vemos que si se acceden múltiples veces a las celdas en los bucles y no se usa un orden lineal se encontrarán fallos en la caché. Para poder solucionar estos problemas se podría reemplazar las matrices bidimensionales por un vector unidimensional plano, esto permitiría aprovechar mejor la caché del procesador.

III.III ESTUDIO DE LA VARIACIÓN DE LA CARGA

En este apartado vamos a estudiar como varía la carga del problema cuando cambiamos el tamaño de este. Para ello, se ha modificado el código para que se pueda analizar el tiempo que tarda dependiendo del tamaño que se le pase como parámetro.

Después de la ejecución de este, obtenemos una tabla donde vemos dependiendo del tamaño, el tiempo que se obtiene y finalmente una media del tiempo de ejecución.

Para este estudio, vemos que principalmente el **parámetro X** que varía la velocidad de nuestro problema son las dimensiones del tablero, lo cual está descrito en el código de la siguiente manera:

```
21      int totalCells;
22      int liveCells;
23      int X_DIM;
24      int Y_DIM;
```

A pesar de ello, también contamos con un **parámetro Y** que puede hacer que la velocidad varie, este es el máximo de generaciones, el cual se describe en el código de la siguiente manera:

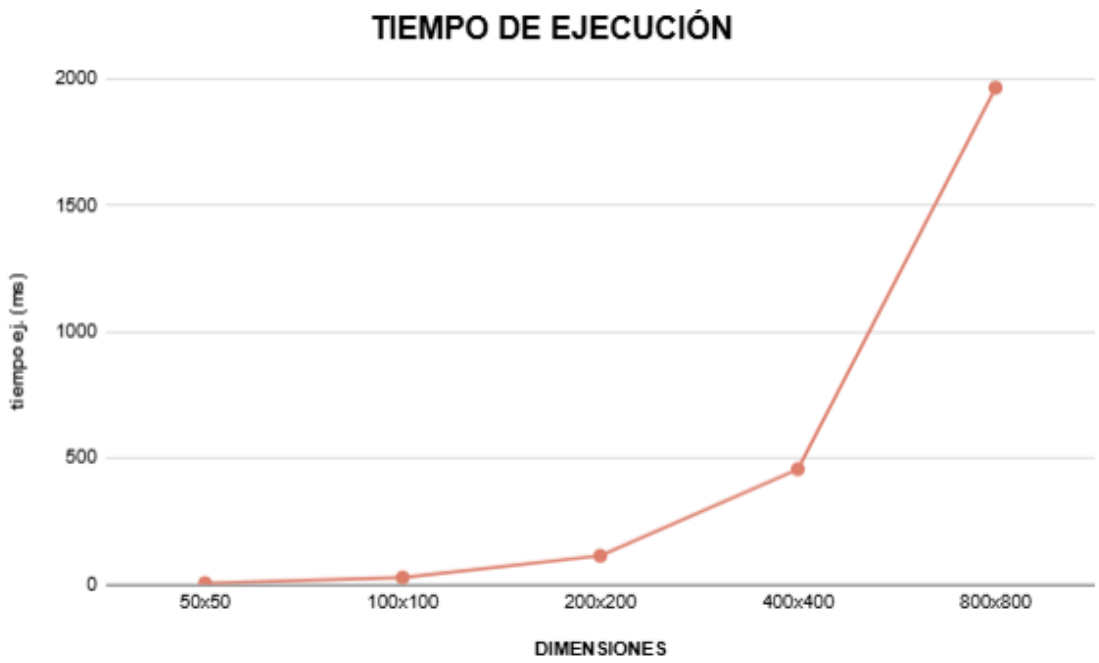
```
22      int liveCells;
23      int X_DIM;
24      int Y_DIM;
25      int MAX_GENS = 1000;
```

Comenzaremos analizando las dos variables de las dimensiones **X_DIM** y **Y_DIM** y en este caso nuestro máximo de generaciones se mantendrá en 1000. Tampoco usaremos ningún parámetro de optimización ya que solo queremos ver como se comportan el problema, estos parámetros serán analizados posteriormente.

Una vez ejecutado nuestro código la salida que se obtiene es la siguiente:

```
g++ -std=c++17 -Wall -c GameOfLife.cpp -o GameOfLife.o
g++ -std=c++17 -Wall -o GameOfLife GameOfLife.o
./GameOfLife -d 5 -fOut output.txt
Size: 50x50     |       Time: 5.17429 ms
Size: 100x100   |       Time: 27.3088 ms
Size: 200x200   |       Time: 113.391 ms
Size: 400x400   |       Time: 456.402 ms
Size: 800x800   |       Time: 1964.53 ms
Average time: 392.906 ms
```

Para visualizar mejor como varía el tiempo se adjunta la siguiente gráfica:



De esta manera podemos ver en el gráfico la relación tiempo dimensión del tablero con una clara tendencia de crecimiento.

Para poder comparar el tiempo entre estos tamaños utilizaremos la siguiente ecuación:

$$relación = \frac{Tactual}{Tmín}$$

En nuestro caso **Tmín** ocupa el tiempo mínimo obtenido en nuestra tabla, en este caso **5,17429**.

Si utilizamos la ecuación descrita anteriormente podemos ver que el paso de una dimensión 50x50 al doble, 100x100, hace que el tiempo de ejecución sea aproximadamente 5 veces el de la dimensión pequeña.

Si por ejemplo comparamos el tiempo de la dimensión 50x50 con al 800x800 podemos ver que este sería 380 veces el de la dimensión pequeña.

A continuación ejecutaremos de nuevo el código pero en este caso cambiando nuestra variable **MAX_GENS** de 1000 a 2000 para ver si se observan cambios en los tiempos de ejecución:

```
./GameOfLife -d 5 -fOut output.txt
Size: 50x50     |       Time: 10.6117 ms
Size: 100x100   |       Time: 52.3999 ms
Size: 200x200   |       Time: 213.842 ms
Size: 400x400   |       Time: 876.303 ms
Size: 800x800   |       Time: 4072.31 ms
Average time: 814.463 ms
```

Para visualizar mejor como varía el tiempo se adjunta la siguiente gráfica:

Al aumentar la variable que indica el máximo de generaciones podemos ver como el tiempo de ejecución se multiplica, pasa de ser 5ms a 10ms el tiempo de la dimensión más pequeña.

Por ello, podemos determinar que este parámetro también es crucial para determinar las variables que hacen variar el tamaño de nuestro problema.

Analizados estos datos podemos ver que en este problema es necesario tomar medidas de paralelización con tal de mejorar el rendimiento.

III.IV "Sintonización" de los parámetros de compilación

En este apartado, como se dijo anteriormente, vamos a estudiar cómo puede variar el tiempo de ejecución de nuestro código dependiendo de los parámetros de optimización que se usen a la hora de compilar.

Para ello, se ha realizado una tabla comparando los distintos parámetros que pueden beneficiar al rendimiento paralelo de nuestro código:

PARÁMETROS DE OPTIMICACIÓN	
PARÁMETRO	**FUNCIÓN**
-O0	En este caso no se realizan optimizaciones, los resultados serían los mismos que los obtenidos en los estudios anteriores.
-O1	Compone uno de los niveles de optimización básica. Tiene como objetivo optimizar la velocidad y evitar el aumento del tamaño del código.
-O2	Compone uno de los niveles de optimización estándar. Permite la vectorización y mejora significativamente el rendimiento.
-O3	Compone uno de los niveles de optimización avanzada. Permite transformaciones en bucle más agresivas. Se recomienda su uso

	para cálculos de punto flotante con muchos bucles y grandes conjuntos de datos.
-march=native	Permite indicar al compilador que genere código máquina optimizado para la CPU en la que se está ejecutando el programa. Esto hace que se habiliten todas las extensiones que soporta la CPU. Su objetivo es maximizar la velocidad y el rendimiento.
-funroll-loops	Permite mejorar el rendimiento aumentando el tamaño del código binario. Principalmente ejecuta múltiples iteraciones de un bucle en una sola. Suele aplicarse por defecto al hacer uso de -O2 y -O3.

A continuación ejecutaremos el código con los distintos parámetros de optimización descritos anteriormente mostrando las salidas obtenidas. En el siguiente apartado se realizará una comparación de estos. El máximo de generaciones usado será de 1000.

CXXFLAGS = -std=c++17 -Wall

```
Size: 50x50     |        Time: 5.27034 ms
Size: 100x100   |        Time: 25.1885 ms
Size: 200x200   |        Time: 108.854 ms
Size: 400x400   |        Time: 440.662 ms
Size: 800x800   |        Time: 1766.13 ms
Average time: 353.226 ms
```

CXXFLAGS = -std=c++17 -Wall **-O1**

```
Size: 50x50     |        Time: 1.34979 ms
Size: 100x100   |        Time: 6.66715 ms
Size: 200x200   |        Time: 29.7501 ms
Size: 400x400   |        Time: 128.807 ms
Size: 800x800   |        Time: 498.386 ms
Average time: 99.6771 ms
```

CXXFLAGS = -std=c++17 -Wall **-O2**

```
Size: 50x50     |        Time: 1.31544 ms
Size: 100x100   |        Time: 6.45635 ms
Size: 200x200   |        Time: 26.3134 ms
Size: 400x400   |        Time: 117.387 ms
Size: 800x800   |        Time: 475.251 ms
Average time: 95.0502 ms
```

CXXFLAGS = -std=c++17 -Wall **-O3**

```
Size: 50x50     |        Time: 0.862031 ms
Size: 100x100   |        Time: 3.85924 ms
Size: 200x200   |        Time: 14.9533 ms
Size: 400x400   |        Time: 66.9255 ms
Size: 800x800   |        Time: 277.908 ms
Average time: 55.5816 ms
```

`CXXFLAGS = -std=c++17 -Wall -O2 -march=native`

```
Size: 50x50     |       Time: 1.37951 ms
Size: 100x100   |       Time: 6.52556 ms
Size: 200x200   |       Time: 26.9781 ms
Size: 400x400   |       Time: 115.62 ms
Size: 800x800   |       Time: 486.749 ms
Average time: 97.3497 ms
```

`CXXFLAGS = -std=c++17 -Wall -O3 -march=native`

```
Size: 50x50     |       Time: 0.820371 ms
Size: 100x100   |       Time: 3.76618 ms
Size: 200x200   |       Time: 14.4948 ms
Size: 400x400   |       Time: 63.2666 ms
Size: 800x800   |       Time: 278.581 ms
Average time: 55.7162 ms
```

`CXXFLAGS = -std=c++17 -Wall -O3 -funroll-loops`

```
Size: 50x50     |       Time: 0.869721 ms
Size: 100x100   |       Time: 3.91118 ms
Size: 200x200   |       Time: 15.1927 ms
Size: 400x400   |       Time: 63.654 ms
Size: 800x800   |       Time: 279.929 ms
Average time: 55.9859 ms
```

III.V OPTIMIZACIÓN

En este apartado realizaremos las comparaciones de los datos obtenidos en el apartado anterior para ver cuales son los parámetros que mejor exprimen la salida que produce nuestro compilador.

A continuación, se mostrará una gráfica en la que se comparan los cuatro principales niveles **-O0**, **-O1**, **-O2** y **-O3**:

Podemos ver que el tiempo de ejecución mejora notablemente cuando pasamos de no usar parámetros de optimización al uso de **-O1**. No obstante, no se observa una gran diferencia entre los tiempos obtenidos con **-O1** que con **-O2**, aunque en este último los son menores.

Cuando si se nota una gran diferencia es del paso de **-O2** a **-O3** ya que el tiempo medio pasa de ser 95 a 55. De este modo, podemos concluir en que el mejor parámetro que se puede usar para optimizar nuestro código es **-O3**.

No obstante, también se han realizado dos pruebas más, por lo que mostraremos una gráfica en la cual se barajan tres posibilidades junto con el parámetro **-O3**:

En la gráfica obtenida no se puede observar gran diferencia entre las tres formas de compilación descritas. Para poder comprarlas se han recogido los datos en la siguiente tabla:

DIMENSIONES	TIEMPO EJ -O3	TIEMPO EJ -march=native	TIEMPO EJ -funroll-loops
50x50	0,861031	0,820371	0,869721
100x100	3,85924	3,76618	3,91118
200x200	14,9533	14,4948	15,1927
400x400	66,9255	63,2666	63,654
800x800	277,908	278,581	279,929

Podemos ver como en dimensiones pequeñas, `-march=native` es una mejor opción frente a utilizar **-O3** como único parámetro. No obstante, cuando las dimensiones son mayores, como la 800x800, resulta no ser tan beneficiosa, esto puede ser debido a que

cuando el problema es grande ese tráfico de memoria se puede volver un cuello de botella al querer procesar varios datos a la vez.

Por otro lado tenemos el parámetro **`-funroll-loops`**, en el cual no se ve ningún tipo de mejora frente a **-O3** ni en dimensiones pequeñas y menos en grandes. Esto puede deberse a que, al ser bucles con pocas iteraciones los que se manejan en este código, no resulta beneficioso desenrollarlos, e incluso empeora como se puede observar el tiempo de ejecución.

Una vez realizados los análisis, la mejor opción para compilar es el uso del parámetro **-O3** ya que se obtiene un beneficio en el tiempo de ejecución independiente de la dimensión y el máximo de generaciones.

III.VI RENDIMIENTO GLOBAL

En este apartado analizaremos los resultados obtenidos en los apartados anteriores sobre el estudio del rendimiento.

Por un lado, analizamos como aumentaba el tiempo de ejecución cuando el tamaño del problema aumentaba. Vimos que esto puede ser debido a dos parámetros:

- Dimensión del tablero
- Máximo de generaciones

En el primer caso, comparamos los tiempos obtenidos con el menor tiempo de la dimensión más pequeña y vimos que este podía llegarse a cuadruplicar. Realizamos una serie de graficas y vimos que había una clara tendencia de crecimiento. Pudimos concluir en que el problema no funcionaba eficientemente con dimensiones grandes.

En el caso de cambiar el máximo de generaciones vimos que si poníamos el máximo a 2000 cuando antes era 1000 todos los tiempos se duplicaban por lo que a mayor máximo de generaciones menos eficiente es nuestro código.

Como se ha comentado en apartados anteriores, este cambio en la eficiencia se debe a como se acceden a los datos en memoria. Ya se ha comentado que las variables calientes son sobre todo **grid** y **neighbors**, al acceder continuamente a estos puede provocar fallos en la caché.

Hay una serie de posibles soluciones, se podría hacer uso de matrices bidimensionales por vectores unidimensionales planos. También se podrían paralelizar los bucles los cuales calculan **nextLiveCells** y **nextGen**.

Por último comentamos que para mejorar el rendimiento se podían hacer eso de parámetros de compilación. Después de realizar el estudio sobre cual era la mejor opción vimos que **-O3** mejoraba notablemente los tiempos de ejecución al ser un nivel de optimización agresiva.

IV. RESOLUCIÓN DE PROBLEMA PROPUESTO (GRIFO)

IV.I PROBLEMA 1.

Un grifo tarda 8 horas en llenar un cierto depósito de agua y otro grifo 40 horas en llenar el mismo depósito. Si usamos los dos grifos para llenar el depósito, que está inicialmente vacío, ¿cuánto tiempo tardaremos? ¿Cuál será la ganancia en velocidad? ¿Y la eficiencia?

Para calcular el tiempo que tardaran estos dos grifos comenzaremos calculando cuánto tarda en llenarse el contenedor por unidad de tiempo:

$$r_1 = \frac{1}{8} \quad r_2 = \frac{1}{40}$$

A continuación, sumamos ambos para saber cuánto tiempo tardarán por unidad de tiempo ambos juntos:

$$r_{total} = r_1 + r_2 = \frac{1}{8} + \frac{1}{40} = \frac{3}{20}$$

finalmente con esta misma relación calculamos el tiempo:

$$r_{total} = \frac{1}{T_{total}} \rightarrow T_{total} = \frac{1}{r_{total}} = \frac{1}{\frac{3}{20}} = 6,\underline{6}h$$

Ahora calcularemos la ganancia de la velocidad:

$$Gp = \frac{T_1}{T_p} \rightarrow \frac{8\ h}{6,\underline{6}h} = 1.2$$

Ahora calcularemos la eficiencia del llenado de los grifos

$$Ep = \frac{Gp}{P} \rightarrow \frac{1.2}{2} = 0.6$$

Esto significa que los grifos aprovechan el 60% del rendimiento teórico, es decir que es muy eficiente.

IV.II PROBLEMA 2.

Suponga que tiene ahora 2 grifos de los que tardan 8 horas en llenar el depósito. Mismas cuestiones que el punto anterior.

Para calcular el tiempo que tardaran estos dos grifos comenzaremos calculando cuánto tarda en llenarse el contenedor por unidad de tiempo:

$$r_1 = \frac{1}{8} \quad r_2 = \frac{1}{8}$$

A continuación, sumamos ambos para saber cuánto tiempo tardarán por unidad de tiempo ambos juntos:

$$r_{total} = r_1 + r_2 = \frac{1}{8} + \frac{1}{8} = \frac{2}{8}$$

finalmente con esta misma relación calculamos el tiempo:

$$r_{total} = \frac{1}{T_{total}} \rightarrow T_{total} = \frac{1}{r_{total}} = \frac{1}{\frac{2}{8}} = 4h$$

Ahora calcularemos la ganancia de la velocidad:

$$Gp = \frac{T_1}{T_p} \rightarrow \frac{8\ h}{4\ h} = 2$$

Ahora calcularemos la eficiencia del llenado de los grifos

$$Ep = \frac{Gp}{P} \rightarrow \frac{2}{2} = 1$$

Esto significa que los grifos utilizan el rendimiento máximo es decir que el sistema no puede ser más eficiente.

IV.III PROBLEMA 3.

Y ahora suponga que tiene 2 grifos de los que tardan 40 horas. Proceda también a realizar los cálculos.

Para calcular el tiempo que tardaran estos dos grifos comenzaremos calculando cuánto tarda en llenarse el contenedor por unidad de tiempo:

$$r_1 = \frac{1}{40} \quad r_2 = \frac{1}{40}$$

A continuación, sumamos ambos para saber cuánto tiempo tardarán por unidad de tiempo ambos juntos:

$$r_{total} = r_1 + r_2 = \frac{1}{40} + \frac{1}{40} = \frac{2}{40}$$

finalmente con esta misma relación calculamos el tiempo:

$$r_{total} = \frac{1}{T_{total}} \rightarrow T_{total} = \frac{1}{r_{total}} = \frac{1}{\frac{2}{40}} = 20h$$

Ahora calcularemos la ganancia de la velocidad:

$$Gp = \frac{T_1}{T_p} \rightarrow \frac{40\ h}{20\ h} = 2$$

Ahora calcularemos la eficiencia del llenado de los grifos

$$Ep = \frac{Gp}{P} \rightarrow \frac{2}{2} = 1$$

Esto significa que los grifos utilizan el rendimiento máximo es decir que este sistema no puede ser más eficiente

IV.III PROBLEMA 4.

Ahora tiene 3 grifos: 2 de los que tardan 40 horas y 1 de los que tardan 8. ¿Qué pasaría ahora?

Para calcular el tiempo que tardaran estos dos grifos comenzaremos calculando cuánto tarda en llenarse el contenedor por unidad de tiempo:

$$r_1 = \frac{1}{40} \quad r_2 = \frac{1}{40} \quad r_2 = \frac{1}{8}$$

A continuación, sumamos ambos para saber cuánto tiempo tardarán por unidad de tiempo ambos juntos:

$$r_{total} = r_1 + r_2 + r_3 = \frac{1}{40} + \frac{1}{40} + \frac{1}{8} = \frac{7}{40}$$

finalmente con esta misma relación calculamos el tiempo:

$$r_{total} = \frac{1}{T_{total}} \rightarrow T_{total} = \frac{1}{r_{total}} = \frac{1}{\frac{7}{40}} = 5.7h$$

Ahora calcularemos la ganancia de la velocidad:

$$Gp = \frac{T_1}{T_p} \rightarrow \frac{8\ h}{5.7\ h} = 1.4$$

Ahora calcularemos la eficiencia del llenado de los grifos

$$Ep = \frac{Gp}{P} \rightarrow \frac{1.4}{3} = 0.46$$

Esto significa que los grifos aprovechan el 46% del rendimiento teórico es decir que no es muy eficiente.

V. CONCLUSIÓN

En esta práctica hemos podido descubrir que tanto la paralelización como la mejora de la estructura de datos aumentan el rendimiento de una aplicación. Gracias al juego *Conway's Game of Life,* se ha podido analizar la estructura del código para poder ver de qué forma podemos paralelizar el código sin alterar el resultado final.

Además, con este juego hemos podido experimentar con sus variables para ver de qué formas podemos mejorar el uso de la lectura y escritura. Llegando a la conclusión de que para aprovechar mejor la caché, es recomendable optimizar la estructura de datos.

Por otro lado, hemos podido concluir que la forma en como compilamos los programas también influye en el rendimiento ya que al compilarlo se crea un archivo en codigo-maquina en el que según las opciones del compilador ordenará las sentencias de forma distinta ayudando a la optimización.

Finalmente, el problema propuesto del llenado del depósito con varios grifos ha servido para entender los conceptos de ganancia en velocidad y eficiencia, comprendiendo que no siempre mejora añadiendo más elementos.

En conclusión, esta práctica ha permitido aplicar los conceptos teóricos de paralelización y optimización para poder decidir el diseño, estructura de datos y configuración del compilador, cosas que influyen directamente en el rendimiento.

VI. REFERENCIAS

AIX. (2025, enero 20). Ibm.com. https://www.ibm.com/docs/es/aix/7.2.0?topic=techniques-compiling-optimization

Mishra, D. (2023, noviembre 30). Optimizaciones del compilador: ¡aumento del rendimiento del código con ajustes mínimos! Hackernoon.com. https://hackernoon.com/lang/es/optimizaciones-del-compilador-aumento-del-rendimiento-del-codigo-con-ajustes-minimos