

UNIVERSIDAD DEL VALLE DE GUATEMALA

Computación Paralela y Distribuida

Sección 10



Excelencia que trasciende

DEL VALLE
GRUPO EDUCATIVO

“Proyecto 1”

ANDREA DE LOURDES LAM PELAEZ 20102

CHRISTOPHER EMANUEL ALEXANDER GARCIA PIXOLA 20541

MARIA ISABEL SOLANO BONILLA 20504

GUATEMALA, Agosto 2023

Índice

Introducción.....	3
Antecedentes.....	4
Programa Secuencial.....	5
Anexo 1.....	5
Diagrama de flujo del programa.....	5
Solicitud de ingreso de datos.....	7
Secuencial.....	7
Paralelo	
Esta parte no tuvo modificaciones, se mantuvo el mismo proceso que en el programa secuencial. Desde línea de comando se solicitó la cantidad de figuras que el usuario desea renderizar. (Ver ejemplo de ejecución en imagen#1).....	7
Captura de argumentos.....	8
Secuencial.....	8
Paralelo.....	8
Esta parte no tuvo modificaciones, se mantuvo el mismo proceso que en el programa secuencial. Desde el método main se realizó la captura de argumentos y asignación a la cantidad de figuras a renderizar en caso fuera un valor válido (Ver ejemplo de ejecución en imagen#2).....	8
Programación defensiva.....	9
Secuencial.....	9
Paralelo.....	9
Se mantuvo la misma programación defensiva en la versión paralela del programa ya que no se realizaron modificaciones en los puntos anteriores. (Ver ejemplo de ejecución en imagen#3).....	9
Secciones paralelas.....	10
Mecanismos de sincronía.....	11
Despliegue de resultados.....	12
Anexo 2.....	14
Catálogo de funciones/variables/estructuras.....	14
Anexo 3.....	15
Pruebas.....	15
- Métricas.....	15
Bitácora.....	15
Conclusiones.....	15
Referencias.....	16

Introducción

En el presente informe se detalla el desarrollo del primer proyecto del curso que consistió en diseñar e implementar un algoritmo secuencial que tuviera alto potencial de ser paralelizable. Posteriormente se buscaría implementar estrategias de mejora y optimización para incrementar el rendimiento del programa. El algoritmo trabajado consistió en un screensaver con algunas restricciones como lo fueron: recibir, por medio de línea de comando, al menos un número n que representara la cantidad de elementos a renderizar en pantalla; Estos objetos debería tener cierta física y alguna característica que reflejara creatividad por parte de los desarrolladores y por último, mostrar los FPS obtenidos tratando de garantizar que este valor no fuera menor a 30. Al no poder utilizar círculos como figura principal, se decidió que trabajaríamos con ondas, estas ondas variarían su tamaño, su orientación, su amplitud y su color; También aparecerían de manera aleatoria a lo largo de todo el canvas en constante movimiento y sin perturbación alguna por la presencia de otra onda en su espacio.

Antecedentes

OpenMP es una herramienta que nos facilita la transformación de un programa secuencial en paralelo, así como nos permite la abstracción y programación paralela en alto nivel. Debido a que está pensado como una solución iterativa, podemos realizar cambios graduales en un programa secuencial para aprovechar múltiples recursos mediante ejecución paralela.

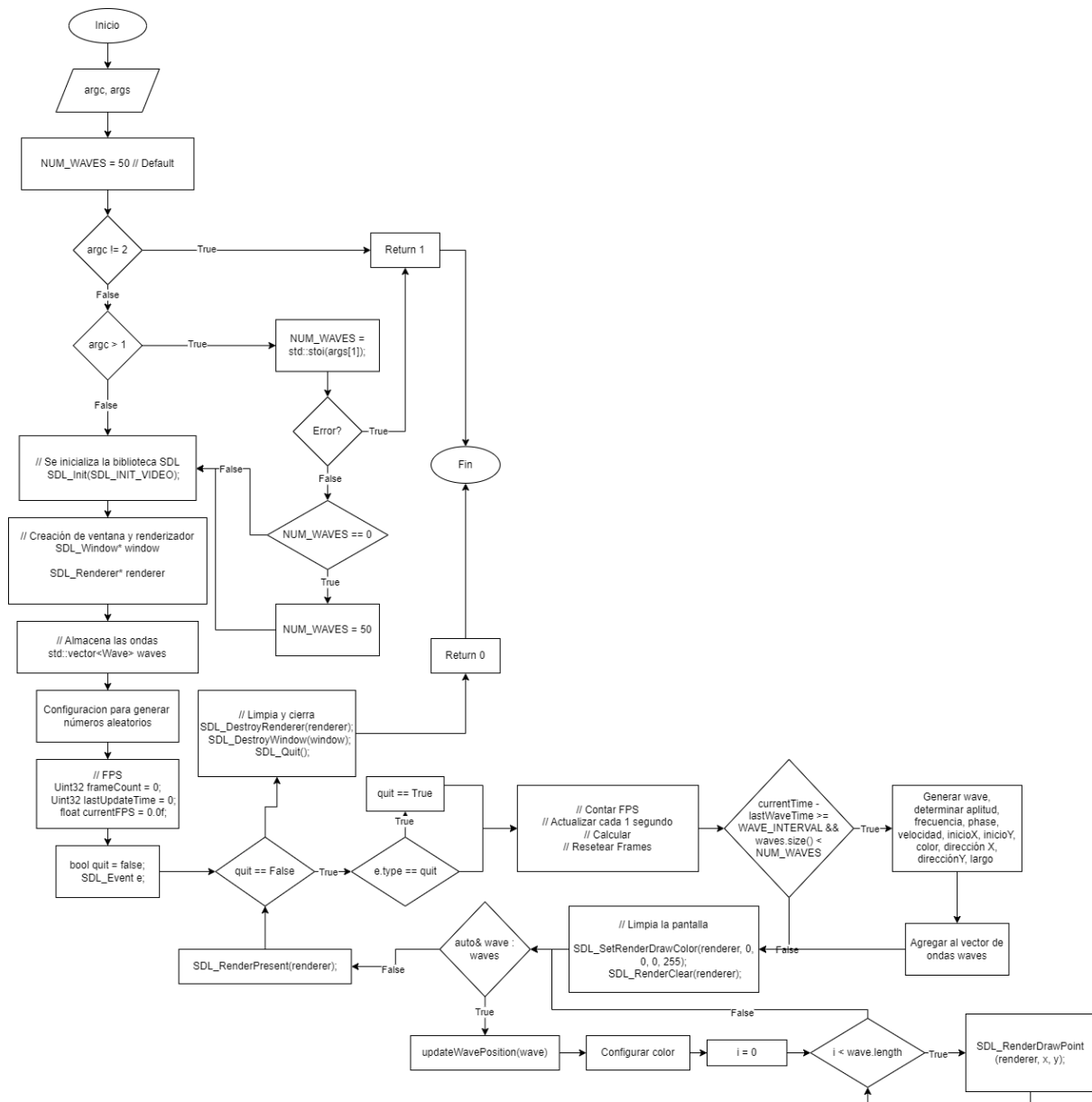
SDL (Lenguaje de Especificación y Descripción) es un lenguaje de modelado formal utilizado para describir sistemas de software y hardware. Fue desarrollado por la Organización Internacional de Normalización (ISO) en 1982. SDL se utiliza para describir sistemas de telecomunicaciones, sistemas de control y sistemas de software en tiempo real. SDL es un lenguaje gráfico que se utiliza para describir el comportamiento de los sistemas. SDL se basa en la teoría de autómatas finitos y se utiliza para describir el comportamiento de los sistemas en términos de estados y transiciones.

En cuanto a los antecedentes, la computación paralela ha sido utilizada históricamente para la simulación de problemas científicos, particularmente en las ciencias naturales e ingeniería, como la meteorología. Esto llevó al diseño de hardware y software paralelo, así como a la computación de alto rendimiento.

Programa Secuencial

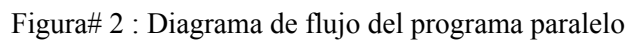
Anexo 1

Diagrama de flujo del programa



Figura# 1 : Diagrama de flujo del programa secuencial

https://drive.google.com/file/d/1AmY-R2M_5q6qdoxwPsq4LENlqv2q4OEh/view?usp=drivesdk



6

Solicitud de ingreso de datos

Secuencial

Para este proyecto la única solicitud de datos que se le hizo al usuario fue la cantidad de figuras a renderizar. Esto se realizó por medio de línea de comando de la siguiente manera:

- **./<nombre del ejecutable> <cantidad de figuras>**

```
christopherg@christopherg-VirtualBox:~/Escritorio/Proyecto#1$ g++ -o Secuencial2 SecuencialV2.cpp -lSDL2
christopherg@christopherg-VirtualBox:~/Escritorio/Proyecto#1$ ./Secuencial2 20
Cantidad de elementos a renderizar: 1: 20
^Cchristopherg@christopherg-VirtualBox:~/Escritorio/Proyecto#1$
```

Imagen#1 : Ejemplo de solicitud de ingreso de datos

Instrucciones de ejemplo:

```
g++ -o Secuencial2 SecuencialV2.cpp -lSDL2
./Secuencialv2 <número de elementos>
```

Paralelo

Esta parte no tuvo modificaciones, se mantuvo el mismo proceso que en el programa secuencial. Desde línea de comando se solicitó la cantidad de figuras que el usuario desea renderizar. (Ver ejemplo de ejecución en imagen#1)

- **./<nombre del ejecutable> <cantidad de figuras>**

```
• andrea@andrea-VirtualBox:~/Documents/GitHub/UVG_Paralela_Proyecto-1$ g++ -o par ParalelaV1.cpp -lSDL2 -fopenmp
• andrea@andrea-VirtualBox:~/Documents/GitHub/UVG_Paralela_Proyecto-1$ ./par 20
Cantidad de elementos a renderizar: 1: 20
FPS: 0
FPS: 0
FPS: 0
```

Imagen #1.1: Ejemplo de solicitud de ingreso de datos

Instrucciones de ejemplo:

```
g++ -o par ParalelaV1.cpp -lSDL2 -fopenmp
./par <num_elementos>
```

Captura de argumentos

Secuencial

Dado que el ingreso de datos se realizó por medio de línea de comando, la forma de capturar argumentos se realizó por medio de la instrucción `argc` y `args` (parámetros del Main), los cuales fueron evaluados con programación defensiva y de esta forma, para este proyecto, se obtuvo la cantidad de figuras a renderizar.

```
int main(int argc, char* args[]) {  
    int NUM_WAVES = 50;  
  
    if (argc != 2) {  
        // Si no se proporciona el número correcto de argumentos, muestra un mensaje de error y salida.  
        std::cout << "Es necesario establecer la cantidad de figuras: ./prog <cantidad>" << std::endl;  
        return 1;  
    }  
  
    if (argc > 1) {  
        try {  
            NUM_WAVES = std::stoi(args[1]);  
            if (NUM_WAVES == 0) {  
                // Si el valor es 0, utiliza el valor predeterminado y muestra un mensaje.  
                NUM_WAVES = 50;  
                std::cout << "Se usará el valor predeterminado de " << NUM_WAVES << std::endl;  
                std::cout << "Cantidad de elementos a renderizar: " << NUM_WAVES << std::endl;  
            } else {  
                for (int i = 1; i < argc; i++) {  
                    std::cout << "Cantidad de elementos a renderizar: " << i << ": " << args[i] << std::endl;  
                }  
            }  
        } catch (std::invalid_argument& e) {  
            std::cout << "Error: Ingreso incorrecto de datos. La cantidad de figuras debe ser un valor numérico." << std::endl;  
            return 1;  
        }  
    }  
}
```

Imagen#2 : Captura de argumentos y programación defensiva en el método main

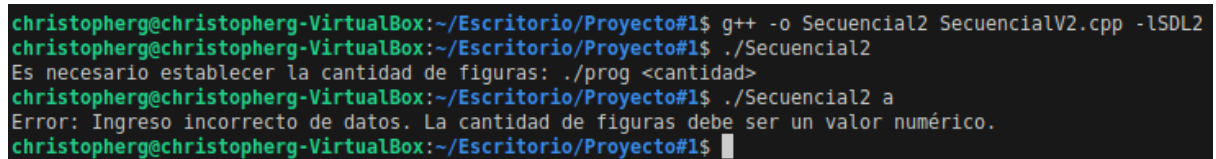
Paralelo

Esta parte no tuvo modificaciones, se mantuvo el mismo proceso que en el programa secuencial. Desde el método main se realizó la captura de argumentos y asignación a la cantidad de figuras a renderizar en caso fuera un valor válido (Ver ejemplo de ejecución en imagen#2)

Programación defensiva

Secuencial

En cuanto a programación defensiva, se trabajó todo a nivel de captura de datos del usuario. Lo primero que se evalúa es que siempre venga el parámetro de cantidad de figuras a renderizar ya que se solicitó que se coloque al menos 1, si esto no sucede el programa no continúa y se muestra en consola un error. El siguiente punto que se evaluó fue el parámetro pasado, este debía cumplir con ser un valor numérico, si esto no se cumple nuevamente se termina el programa y se muestra un error en consola. Como grupo decidimos tener un valor default al que el usuario podrá acceder si coloca 0 como parámetro, esto se dejó así para tener siempre una manera de correr el screensaver si no se sabe con claridad la cantidad de figuras que se desea mostrar.



```
christopherg@christopherg-VirtualBox:~/Escritorio/Proyecto#1$ g++ -o Secuencial2 SecuencialV2.cpp -lSDL2
christopherg@christopherg-VirtualBox:~/Escritorio/Proyecto#1$ ./Secuencial2
Es necesario establecer la cantidad de figuras: ./prog <cantidad>
christopherg@christopherg-VirtualBox:~/Escritorio/Proyecto#1$ ./Secuencial2 a
Error: Ingreso incorrecto de datos. La cantidad de figuras debe ser un valor numérico.
christopherg@christopherg-VirtualBox:~/Escritorio/Proyecto#1$
```

Imagen#3 : Mensajes de advertencia y error por ingreso incorrecto de datosR

Paralelo

Se mantuvo la misma programación defensiva en la versión paralela del programa ya que no se realizaron modificaciones en los puntos anteriores. (Ver ejemplo de ejecución en imagen#3)

Secciones paralelas

```
// Limpia la pantalla
SDL_SetRenderDrawColor(renderer, 0, 0, 0, 255);
SDL_RenderClear(renderer);

for (auto& wave : waves) {
    updateWavePosition(wave);

    // Configura el color de la onda
    SDL_SetRenderDrawColor(renderer, (wave.color >> 24) & 0xFF, (wave.color >> 16) & 0xFF, (wave.color >> 8) & 0xFF, (wave.color >> 0) & 0xFF);

    for (int i = 0; i < wave.length; ++i) {
        // Dibuja puntos que forman la onda en movimiento
        int x = wave.startX + static_cast<int>(i * wave.directionX);
        int y = wave.startY + static_cast<int>(i * wave.directionY + wave.amplitude * sin(2 * M_PI * i / wave.length));
        SDL_RenderDrawPoint(renderer, x, y);
    }
}

// Renderiza la escena
SDL_RenderPresent(renderer);
}
```

Imagen #4: Programa antes de paralelización

```
// Limpia la pantalla
SDL_SetRenderDrawColor(renderer, 0, 0, 0, 255);
SDL_RenderClear(renderer);

omp_init_lock(&wavesMutex); // se inicializa el mutex

#pragma omp parallel for
for (size_t i = 0; i < waves.size(); ++i) {
    Wave& wave = waves[i]; // cada thread trabaja en una onda distinta

    // Actualiza la posición de la onda
    updateWavePosition(wave);

    omp_set_lock(&wavesMutex);

    // Configura el color de la onda
    SDL_SetRenderDrawColor(renderer, (wave.color >> 24) & 0xFF, (wave.color >> 16) & 0xFF, (wave.color >> 8) & 0xFF, (wave.color >> 0) & 0xFF);

    for (int i = 0; i < wave.length; ++i) {
        // Dibuja puntos que forman la onda en movimiento
        int x = wave.startX + static_cast<int>(i * wave.directionX);
        int y = wave.startY + static_cast<int>(i * wave.directionY + wave.amplitude * sin(2 * M_PI * i / wave.length));
        SDL_RenderDrawPoint(renderer, x, y);
    }

    omp_unset_lock(&wavesMutex);
}

// Renderiza la escena
SDL_RenderPresent(renderer);
}
```

Imagen #5 Programa después de agregarle los cambios para que sea paralelo

En esta sección se usa “#pragma omp parallel for” para distribuir el trabajo de procesamiento de las ondas entre múltiples hilos. Cada hilo trabaja una onda distinta del vector “waves”.

Las tareas que se realizan de manera paralela son:

- Actualización de la posición de la onda: Cada hilo actualiza la posición de la onda llamando a la función “updateWavePosition(wave)”
- Configuración del color de la onda: Cada hilo configura el color de la onda para el dibujo utilizando la función “SDL_SetRenderDrawColor.”
- Dibujo de puntos que forman la onda en movimiento: Cada hilo itera sobre los puntos que forman la onda y dibuja los puntos en la pantalla utilizando la función “SDL_RenderDrawPoint”

Mecanismos de sincronía

- Mutex (omp_lock_t)

```
std::vector<Wave> waves; // Almacena las ondas
omp_lock_t wavesMutex; // Se inicializa mutex
```

Imagen #6: Implementación de Mutex para sincronización

Este mutex se utiliza para asegurar que solo un hilo a la vez pueda acceder a ciertas partes críticas del código. En particular, se utiliza para proteger el acceso concurrente al vector waves y a la configuración del color de la onda dentro del bucle paralelo.

- Locks (omp_set_lock y omp_unset_lock):

```
// Actualiza la posición de la onda
updateWavePosition(wave);

omp_set_lock(&wavesMutex);
```

Imagen #7: Implementación de set de locks para mecanismos de sincronización

```
    SDL_RenderDrawPoint(renderer, x, y
}

omp_unset_lock(&wavesMutex);
}

// Renderiza la escena
SDL_RenderPresent(renderer);
```

Imagen #8 : Implementación de unset de locks para mecanismos de sincronización

Dentro del bucle paralelo, se utilizan las funciones `omp_set_lock` y `omp_unset_lock` para adquirir y liberar el mutex respectivamente. Estas llamadas se realizan antes y después de las operaciones críticas para garantizar que un hilo a la vez tenga acceso a las partes críticas del código. Estas llamadas a funciones aseguran que un hilo no acceda a waves o a la configuración de color mientras otro hilo está trabajando en la misma parte del código, evitando así problemas de concurrencia.

Despliegue de resultados

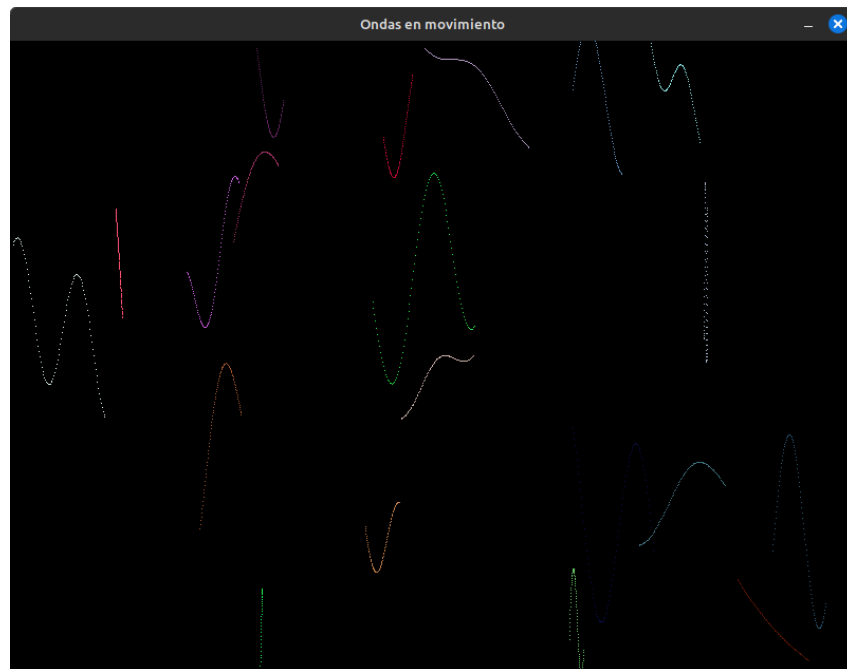


Imagen #9: Resultado en pantalla de programa secuencial

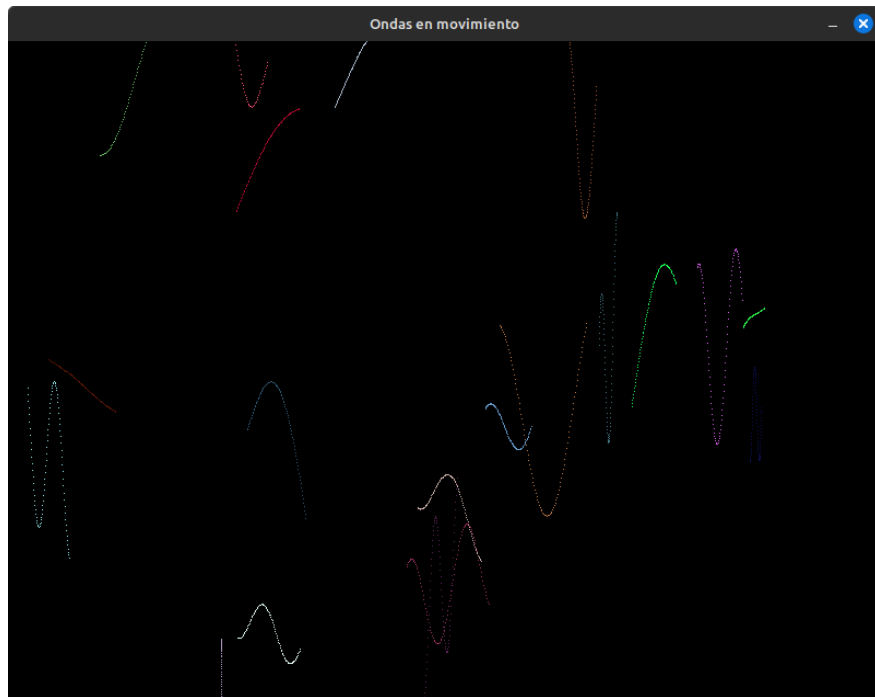


Imagen #10: Visualización de pantalla en programa paralelo

Anexo 2

Catálogo de funciones/variables/estructuras

Nombre	Entradas	Salidas	Descripción
updateWavePosition	Wave& wave	N/A	Actualiza la posición de la onda permitiendo la simulación de movimiento.
generateRandomColor	Wave& wave	N/A	Genera un color RGB aleatorio que se le asigna a la onda como propiedad.
Wave	N/A	N/A	Estructura de las ondas. Estas poseen: amplitud, frecuencia, fase, velocidad, posición inicial en X, posición inicial en Y, dirección en X, dirección en Y, color y longitud.
SCREEN_WIDTH	N/A	N/A	Ancho de la pantalla
SCREEN_HEIGHT	N/A	N/A	Alto de la pantalla
WAVE_INTERVAL	N/A	N/A	Intervalo de tiempo entre generación de ondas
INITIAL_WAVE_LENGTH	N/A	N/A	Longitud inicial de las ondas
PI	N/A	N/A	Valor de PI
NUM_WAVES	N/A	N/A	Cantidad de ondas a renderizar

Anexo 3

Pruebas

- Métricas

- Speedup:

$$Speedup = \frac{T. Secuencial}{T. Paralelo}$$

- Efficiency:

$$Efficiency = \frac{speedup}{N. hilos} * 100 = \%$$

Bitácora

Cantidad de Elementos	FPS promedio		Speedup	eficiencia
	Secuencial	Paralela		
10	898.137	126.6	7.094	70.94
20	800.039	179.81	4.449	22.25
30	781.831	167.6	4.664	15.55
40	741.015	115.2	6.432	16.08
50	675.116	132.7	5.087	10.18
60	608.028	125.5	4.844	8.07
70	526.183	182.1	2.889	4.13
80	514.59	155.7	3.305	4.03
90	492.243	143.3	3.435	3.82

Discusión y Conclusiones

Como se puede observar en los resultados obtenidos, en el caso específico del programa desarrollado para el proyecto, la paralelización no ayudó a este programa para mejorar la renderización de los frames por segundo. Esto tiene mucha lógica, ya que por el uso de mutex en lugar de permitir que los threads modifiquen la memoria en cualquier momento con la posibilidad de sobre escribir y generar *racing conditions*, los mutex obligan a que los threads esperen a que se escriba a memoria. Esto a su vez genera que haya un delay y este se ve reflejado en la renderización del programa paralelo. A pesar de las diversas pruebas que se hicieron para buscar mejorar el rendimiento del programa paralelo; usando distintas directivas, agregando más áreas paralelas, determinando distintas cantidades de threads, etc, no se logró a que este mejorara. Otra de la razones por la que el programa paralelo pudo presentar resultados no tan favorables puede ser la sobrecarga de paralelización provocando que el programa en cuestión perdiera algunos FPS durante el proceso de renderización.

Referencias

- Grahlmann, B. (1998). Combining finite automata, parallel programs and SDL using Petri nets. In Tools and Algorithms for the Construction and Analysis of Systems (pp. 102-117). Springer.
- Shen, H., & Zhang, Y. (2013). Comparison and Analysis of Parallel Computing Performance Using OpenMP and MPI. The Open Automation and Control Systems Journal, 5, 38-44.
<https://doi.org/10.2174/1874444301305010038>
- Singh, S., & Kumar, A. (2022). Parallel implementation of solving linear equations using OpenMP. Journal of Computational Electronics. Advance online publication.
<https://doi.org/10.1007/s41870-022-00899-9>
- Kozłowski, M., & Kierzyńska, M. (2015). The Feasibility of Using OpenCL Instead of OpenMP for Parallel CPU Computing. arXiv preprint arXiv:1503.06532.