[PROYECTO #1]

[Computación Paralela y Distribuida]

José Auyón 201579
Marco Ramirez 21032
Josué Morales 21116

Link de Github: https://github.com/mvrcentes/Computacion_Paralela_2024.git

Resumen

Este proyecto busca el diseño, implementación y optimización de un screensaver utilizando técnicas de computación paralela con OpenMP. Inicialmente, se desarrolló una versión secuencial del screensaver que muestra la interacción dinámica de múltiples elementos gráficos en movimiento. Posteriormente, se aplicaron estrategias de paralelización para mejorar el rendimiento y la eficiencia del programa, logrando un aumento significativo en los FPS y una utilización más efectiva de los recursos del sistema. Los resultados demuestran que la aplicación de paralelismo en tareas gráficas puede potenciar el desempeño y ofrecer una experiencia visual más fluida y atractiva.

Introducción

En la actualidad, la necesidad de procesar y renderizar datos de manera eficiente es muy importante, especialmente en aplicaciones gráficas y de simulación. La computación paralela es una solución efectiva para aprovechar al máximo los recursos de hardware disponibles, permitiendo la ejecución simultánea de varias tareas y optimizando el tiempo de procesamiento.

OpenMP es una librería que ha sido utilizada para la programación paralela en arquitecturas de memoria compartida. Por facilitar la transformación de programas secuenciales en paralelos mediante directivas simples, permitiendo a los desarrolladores mejorar el rendimiento de sus aplicaciones de manera gradual y controlada.

Este proyecto tiene como objetivo aplicar los conceptos de computación paralela en el contexto del desarrollo de un screensaver interactivo primero secuencial y luego paralelo. Mediante el uso de OpenMP y la librería gráfica SDL, se busca crear una aplicación que use la paralelización para mejorar el rendimiento secuencial del Screensaver.

Metodología

El desarrollo del proyecto se llevó a cabo en varias etapas, siguiendo una metodología estructurada que facilitó la implementación y optimización del screensaver.

1. Diseño Conceptual del Screensaver

Idea Principal: Se planteó un screensaver que muestra una cantidad N de círculos luminosos que se mueven de forma aleatoria dentro de un espacio, interactuando entre sí y con los bordes de la pantalla mediante choques elásticos.

Elementos Gráficos: Los círculos presentan colores variados generados aleatoriamente, creando un efecto visual de movimiento y choques.

2. Implementación de la Versión Secuencial

Entorno de Desarrollo: Se utilizó el lenguaje de programación C++ junto con la librería gráfica SDL2 para el renderizado de los elementos en pantalla.

Estructura del Código:

Inicialización: Configuración de la ventana gráfica con una resolución de 800x600 píxeles y preparación del entorno SDL.

Generación de Partículas: Creación de N círculos con posiciones, velocidades y colores iniciales asignados aleatoriamente.

Bucle Principal: Ciclo que actualiza la posición de los círculos, detecta y maneja colisiones, y renderiza el estado actual en cada frame.

Cálculo de FPS: Implementación de un contador que calcula y muestra en pantalla los frames por segundo.

3. Análisis de Puntos de Paralelización

Identificación de Tareas Paralelas: Se analizaron las partes del código que podrían paralelizarse, llegando a la conclusión de que son:

- El cálculo de nuevas posiciones y detección de colisiones.
- La creación de cada partícula en pantalla.

Se evaluaron los posibles problemas de race conditions y se planificaron mecanismos de sincronización adecuados para asegurar la coherencia de los datos.

4. Implementación con OpenMP

Paralelización de la Actualización de Partículas:

Se aplicaron directivas #pragma omp parallel for con schedule(dynamic) en los bucles que actualizan las posiciones y manejan las colisiones de las partículas, lo que permitió distribuir dinámicamente la carga de trabajo entre los hilos.

Paralelización del Renderizado:

Se implementó la paralelización en el proceso de dibujado de las partículas, utilizando múltiples hilos para reducir el tiempo de renderizado por frame.

Ajuste de Número de Hilos:

Se experimentó con diferentes números de hilos para encontrar el equilibrio entre rendimiento y utilización de recursos.

Mecanismos de Sincronización:

Se utilizaron directivas como #pragma omp critical para proteger secciones donde se accede a datos compartidos (*Z/OS 2.4.0*, s. f.), #pragma omp atomic para actualizar variables de forma segura con mínima sobrecarga (*XL C/C++ For Linux 16.1.0*, s. f.), y #pragma omp sections para ejecutar diferentes tareas en paralelo (*Z/OS 2.4.0*, s. f.-b). Además, se implementaron barreras de sincronización para asegurar que todos los hilos completen sus tareas antes de avanzar a la siguiente fase.

5. Pruebas y Evaluación de Rendimiento

Escenarios de Prueba:

Se realizaron pruebas con diferentes valores de N (número de círculos) para evaluar el rendimiento bajo distintas cargas de trabajo.

Se compararon los resultados entre la versión secuencial y la versión paralela en términos de FPS y uso de CPU.

6. Recopilación de Datos:

Se registraron métricas de rendimiento, incluyendo el tiempo promedio de procesamiento por frame y la eficiencia de la paralelización.

7. Análisis de Resultados:

Se analizaron los datos recopilados para identificar mejoras y posibles cuellos de botella en el rendimiento.

Se realizaron ajustes y optimizaciones adicionales basadas en los resultados de las pruebas.

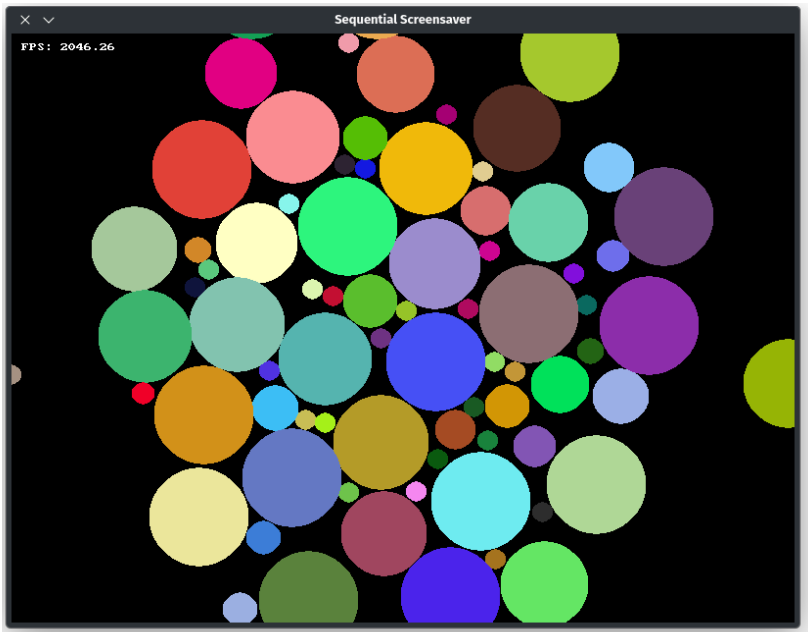
Resultados

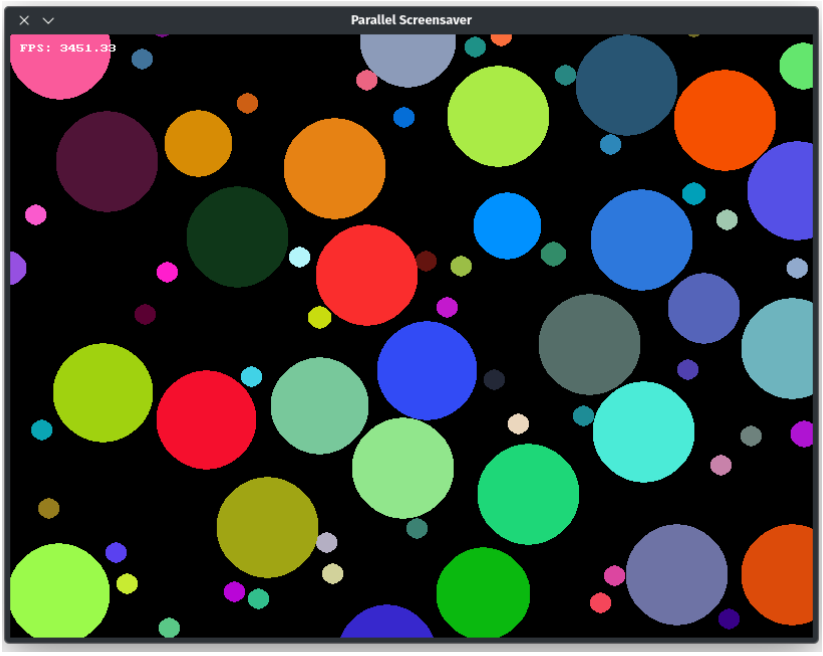
Los resultados obtenidos tras la implementación y optimización del screensaver reflejan mejoras significativas en el rendimiento y la eficiencia del programa

Cantidad de círculos	FPS Secuencial	FPS Paralelo	% de mejora
100	2046.26	3451.33	68.67%
500	438.93	459.37	4.65%
1000	118.57	163.39	37.79%
1200	82.94	144.66	74.41%
2000	33.18	73.02	119.99%

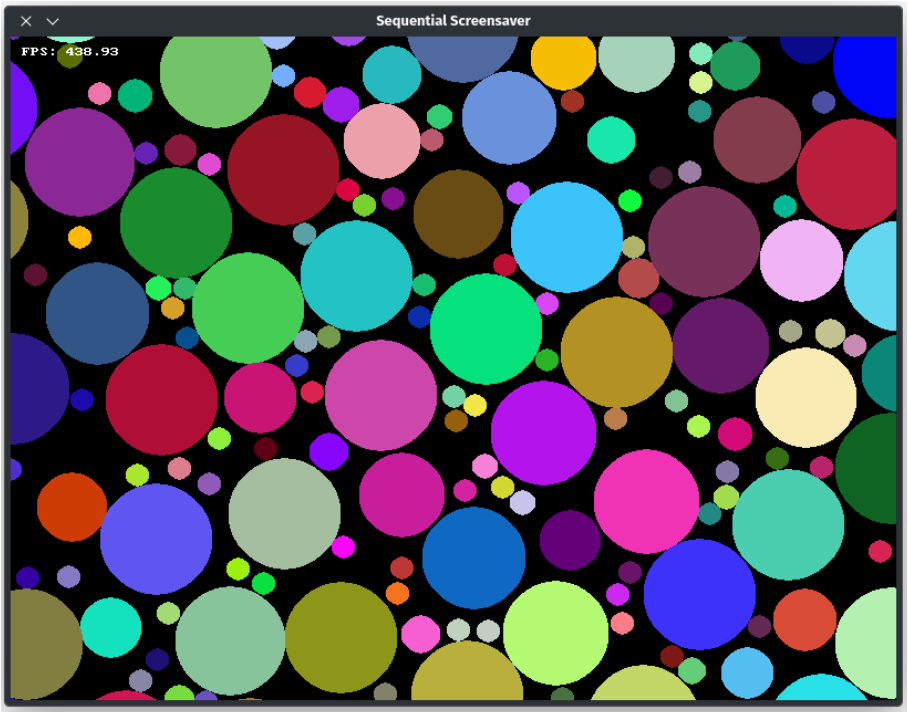
Pruebas:

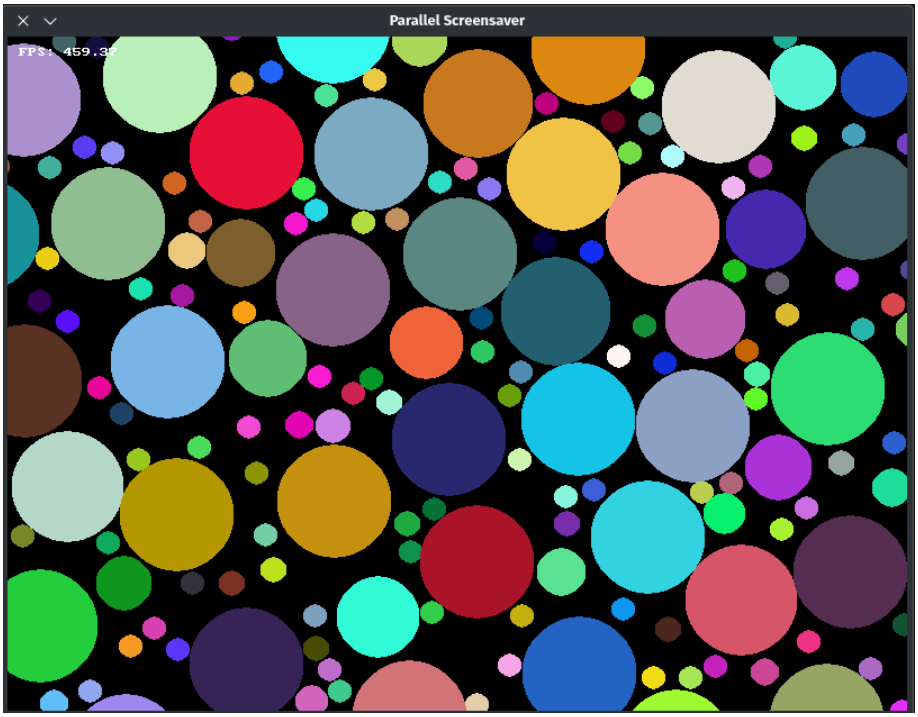
N = 100



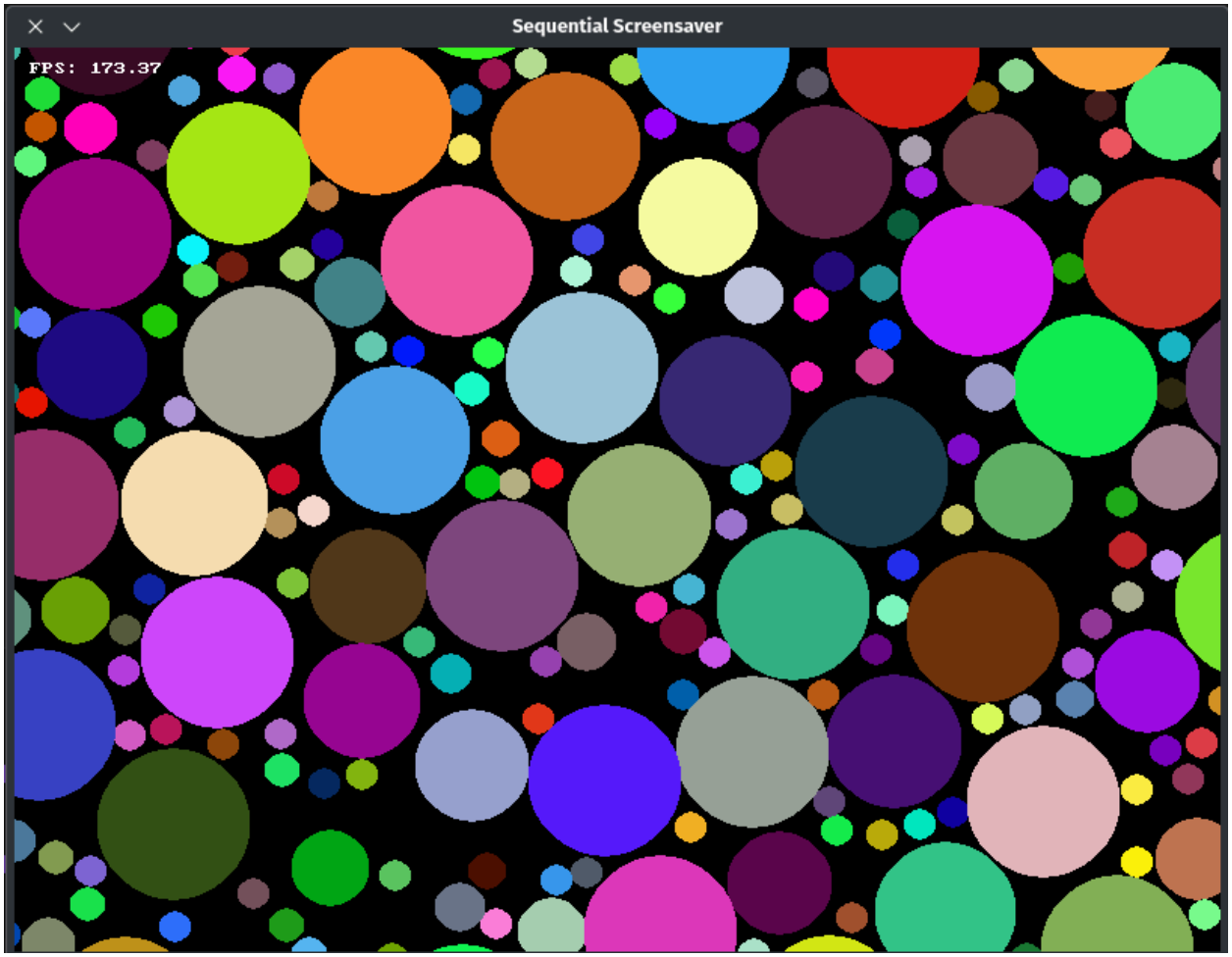


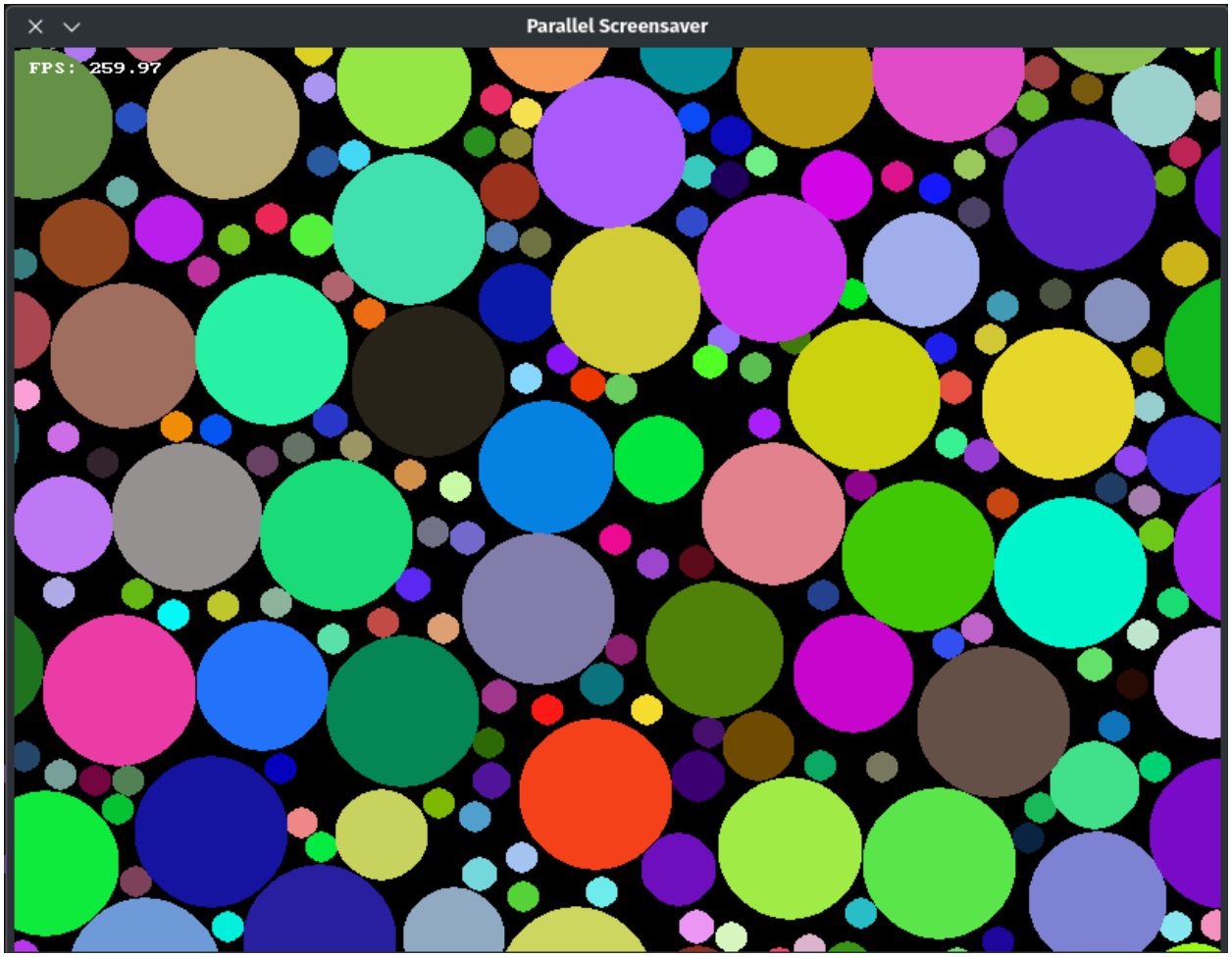
N = 500



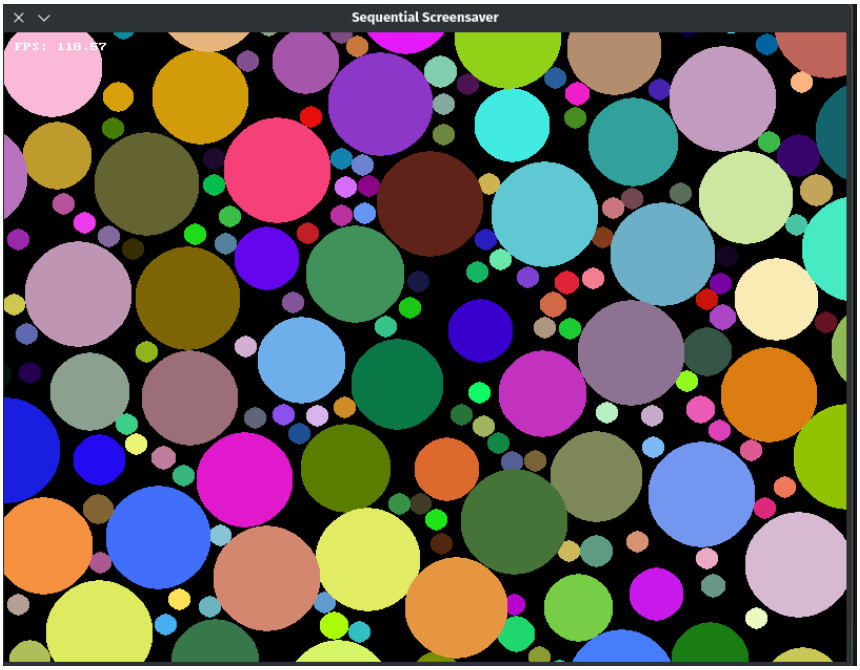


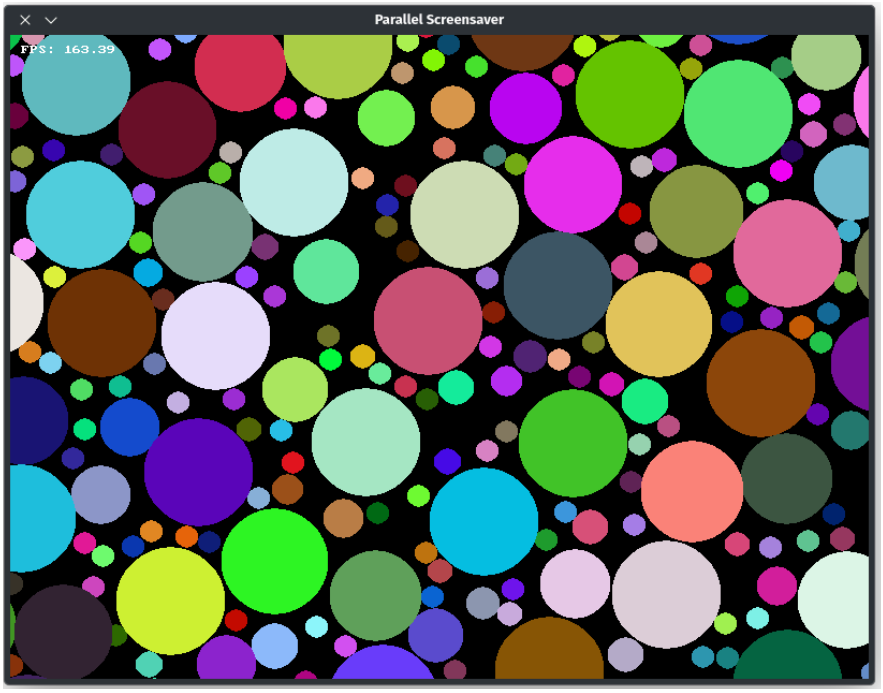
N= 800


Sequential Screensaver
FPS: 173.37

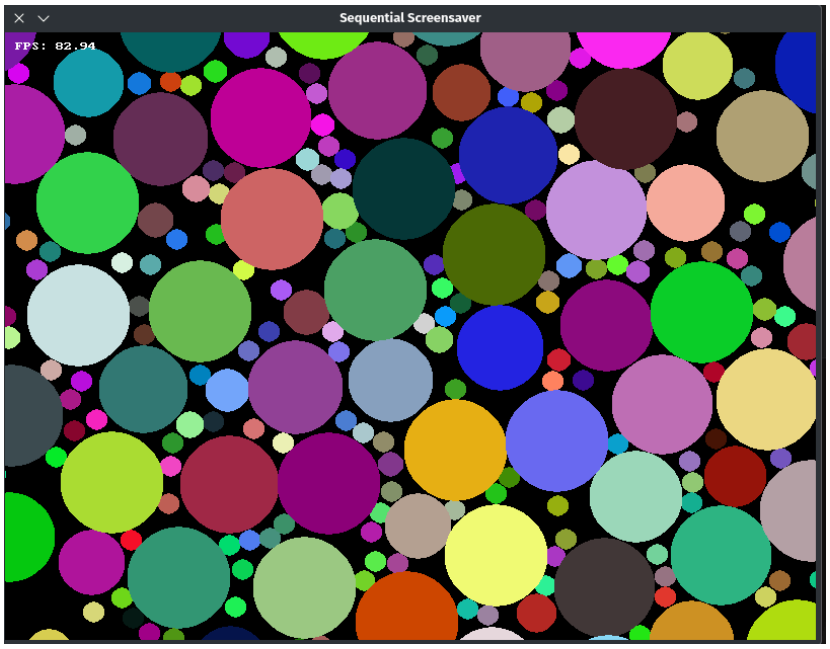


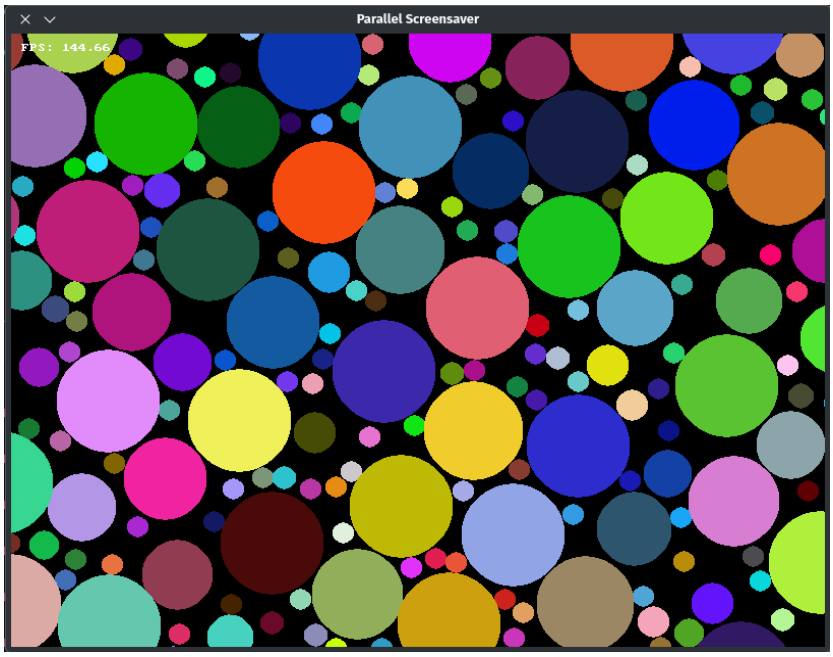
N = 1000


Sequential Screensaver
FPS: 118.77

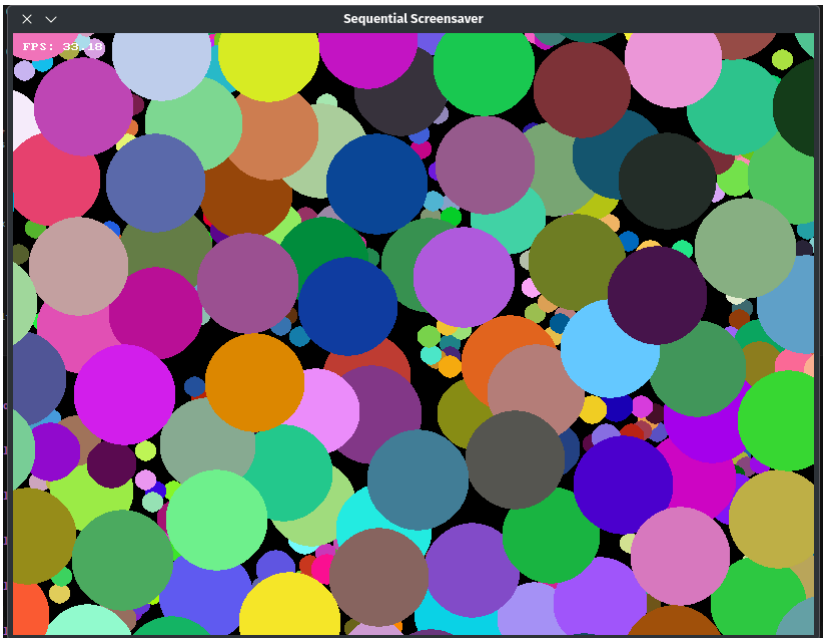

Parallel Screensaver
FPS: 163.39

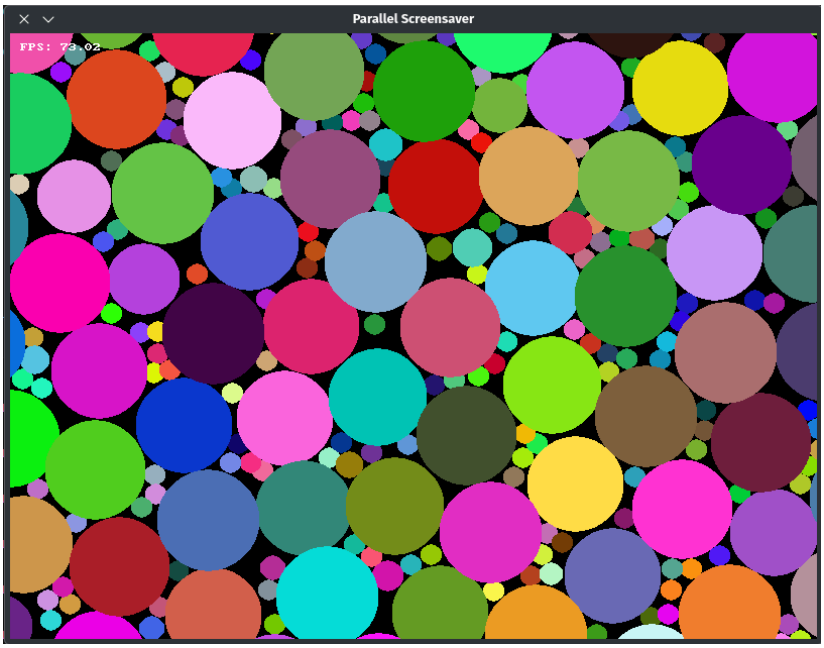
N = 1200


Sequential Screensaver
FPS: 82.94

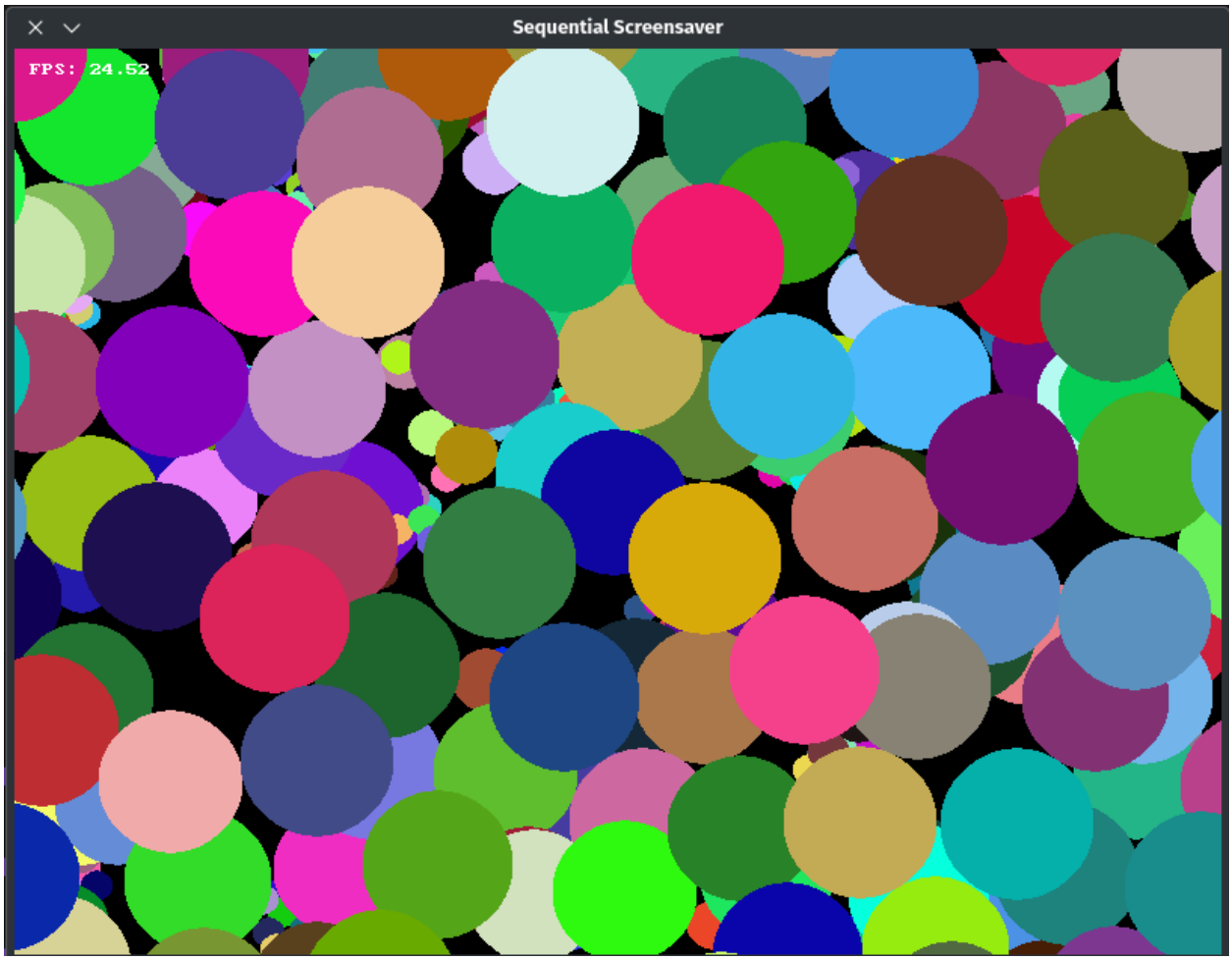

Parallel Screensaver
FPS: 144.66

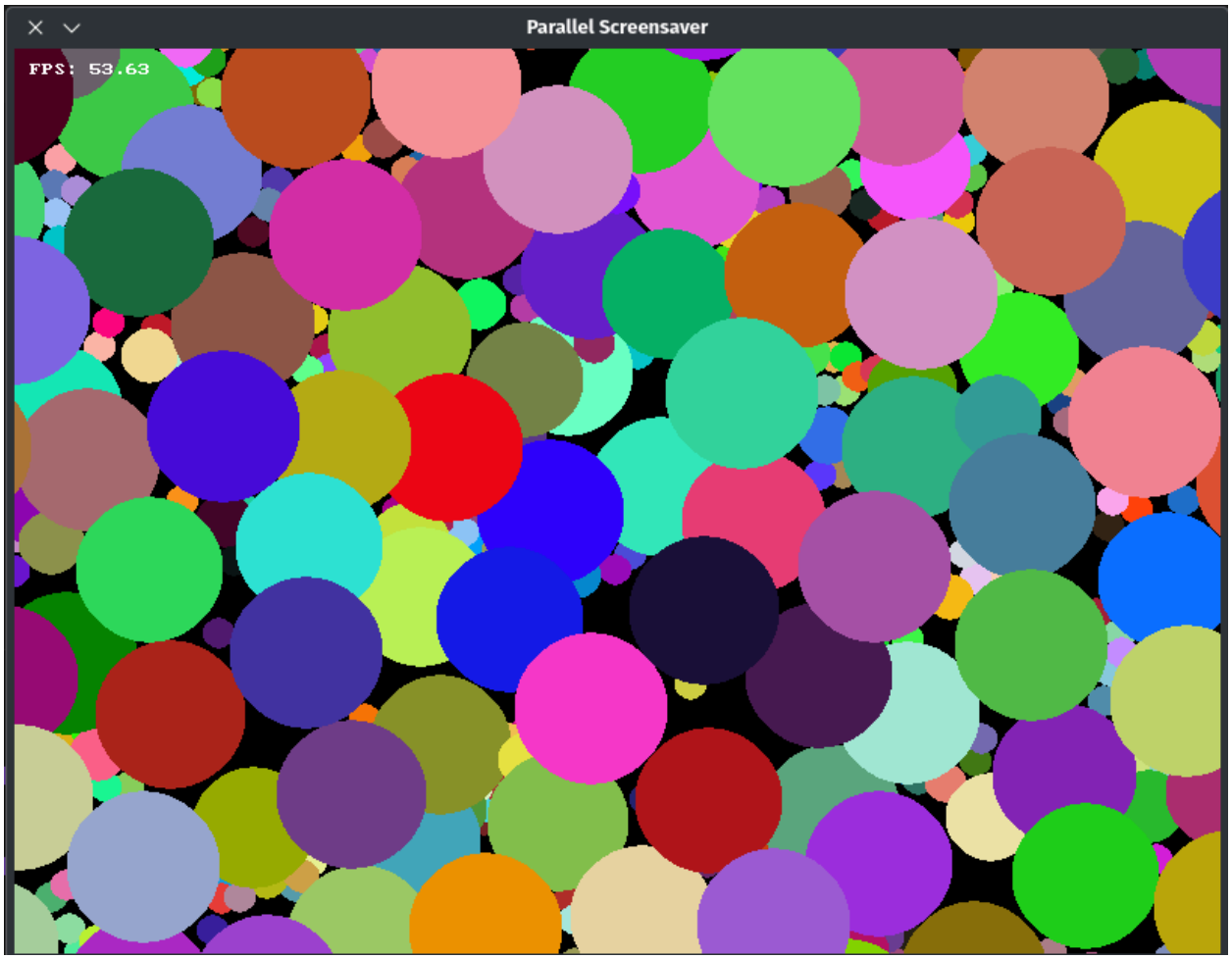
N = 2000


Sequential Screensaver
FPS: 33.18

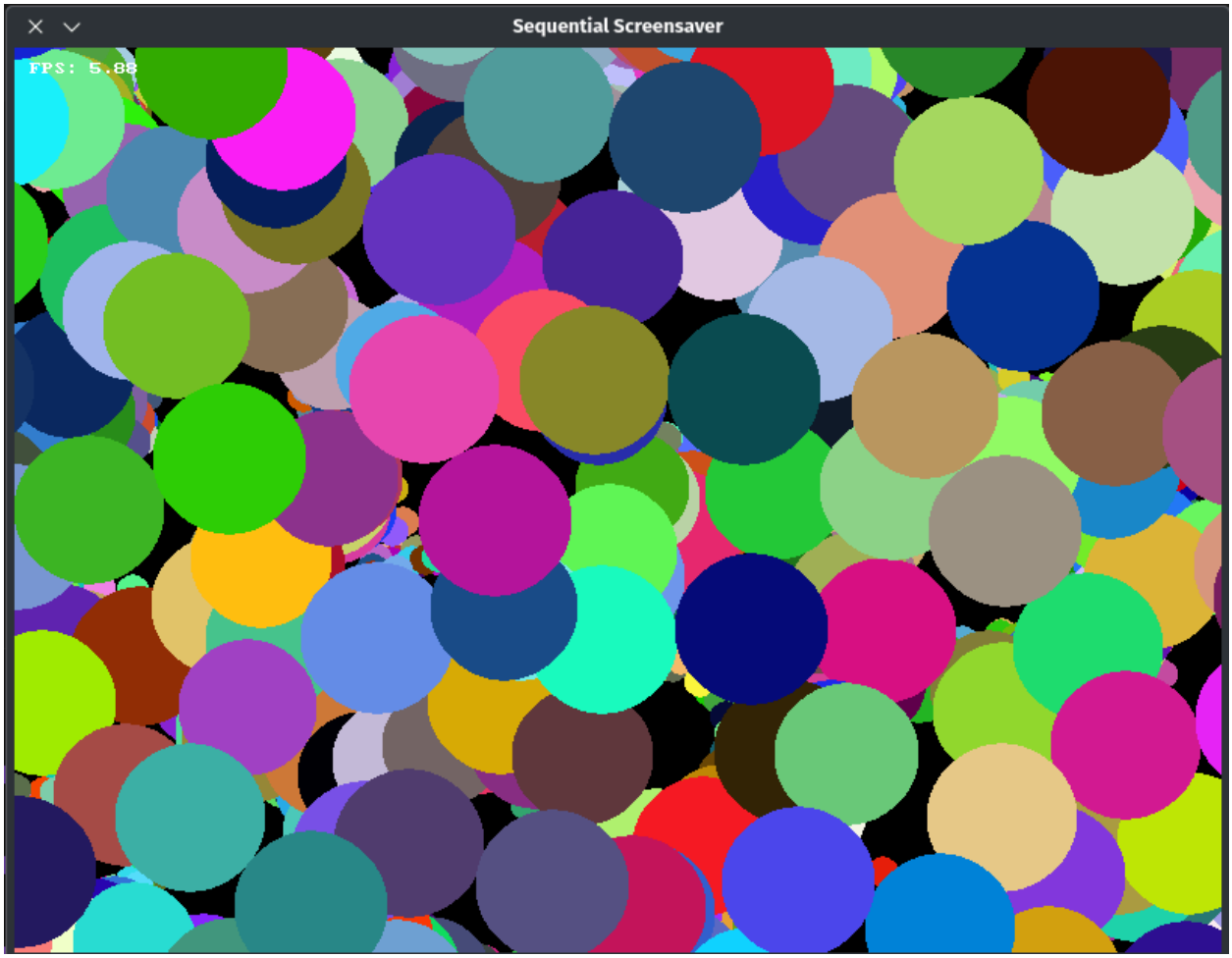

Parallel Screensaver
FPS: 73.02

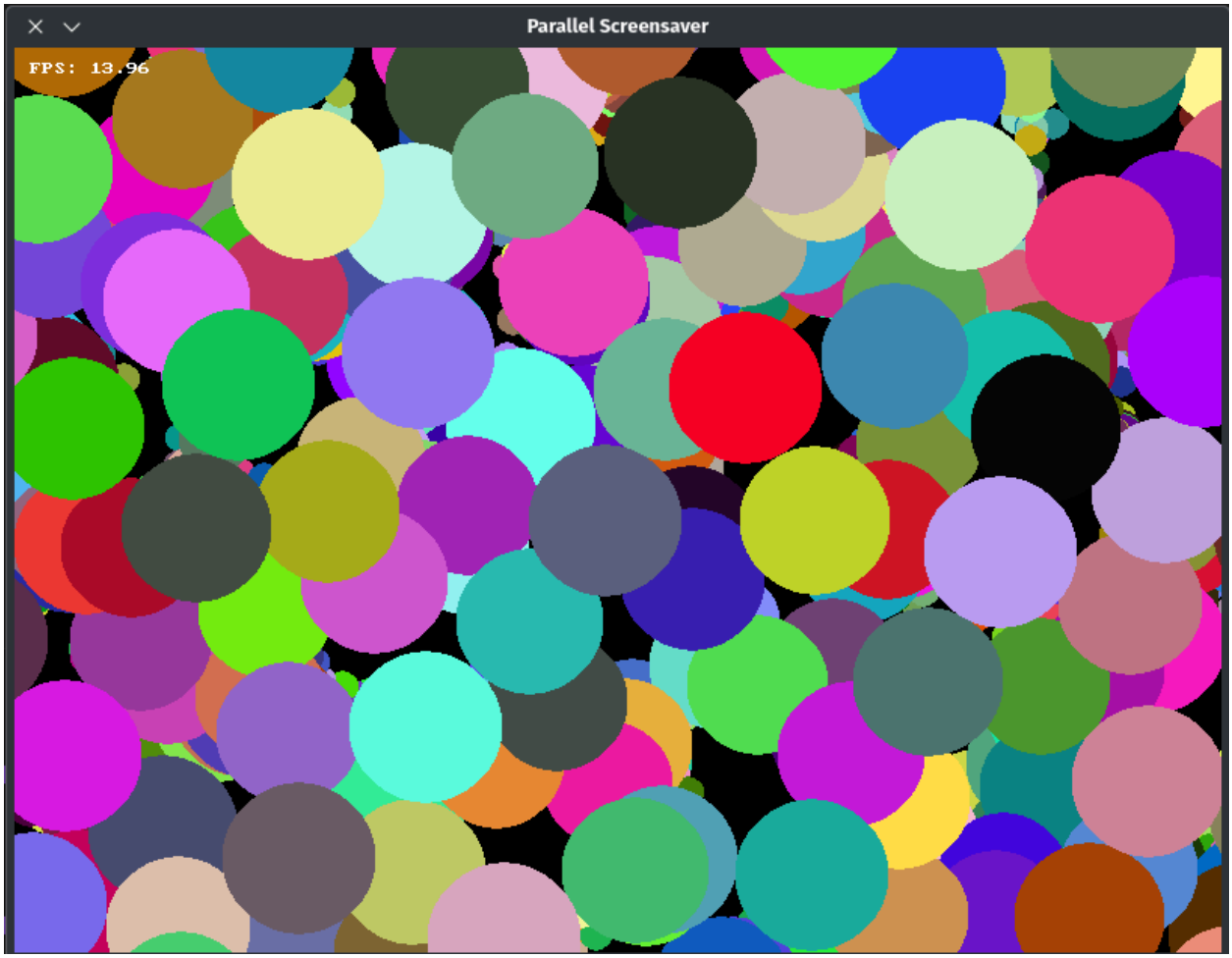
N=2500



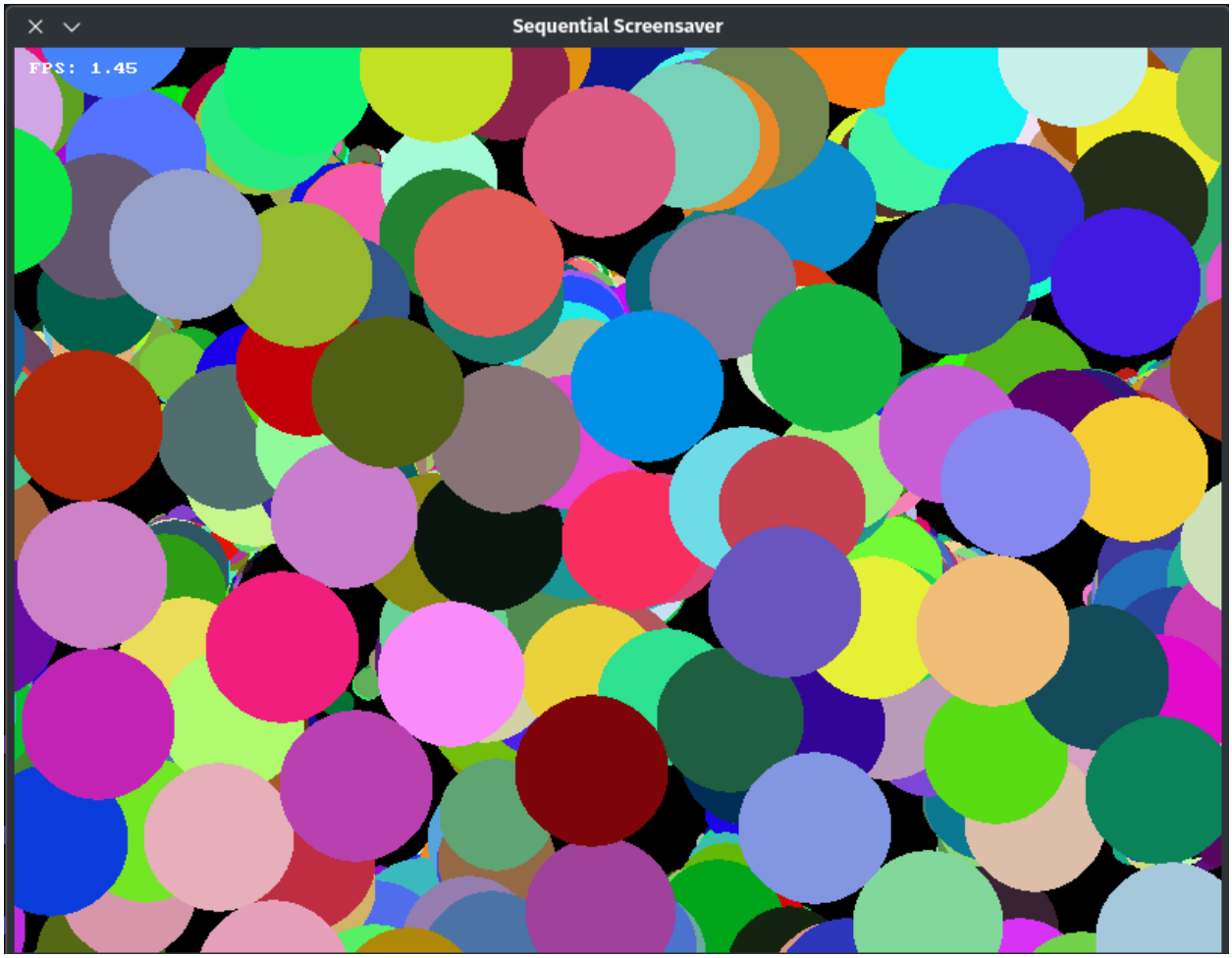


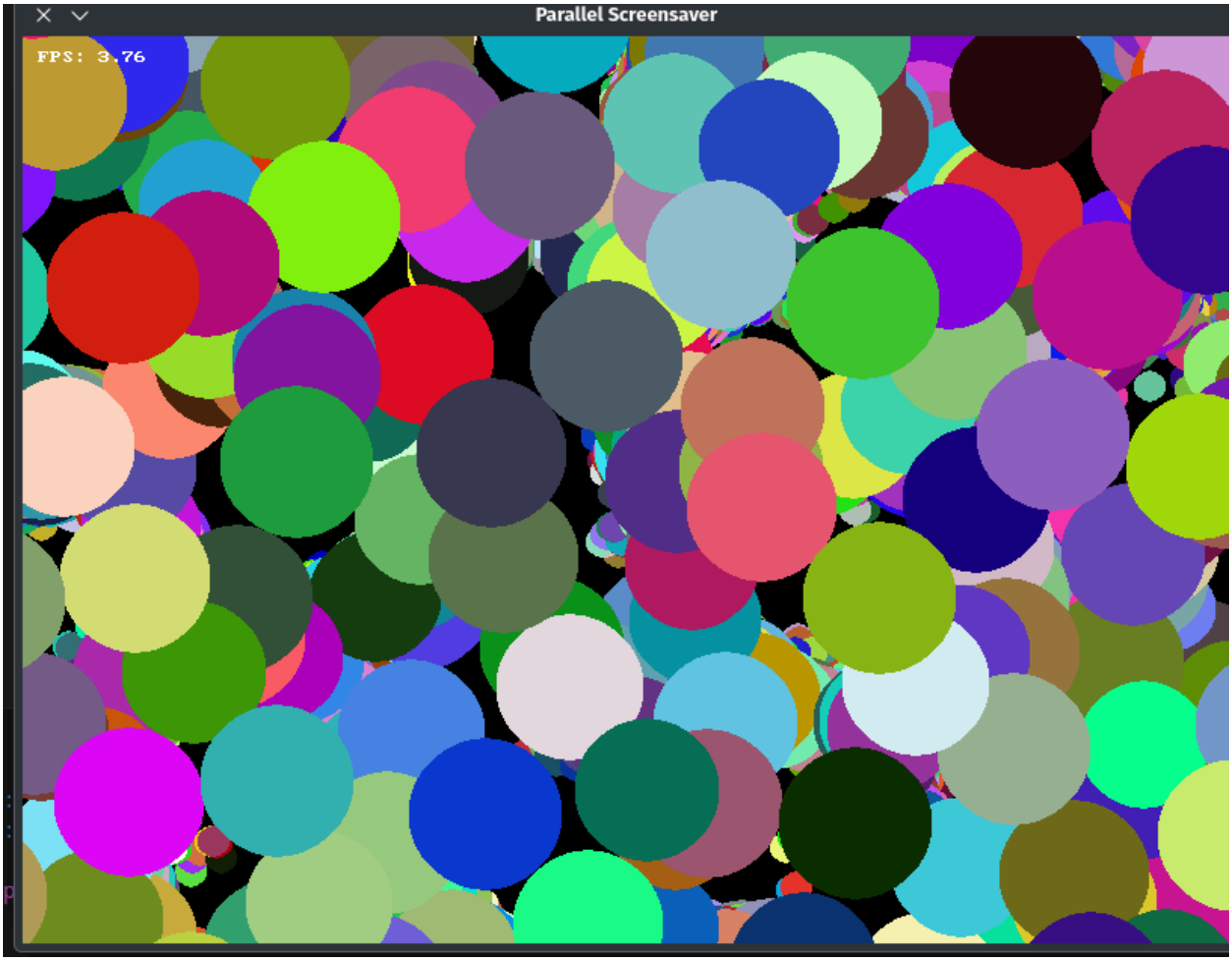
N = 5000


Sequential Screensaver
FPS: 5.88



N = 10,000


Sequential Screensaver
FPS: 1.45



Discusión

Paralelizar la actualización y el renderizado de los círculos, mejoró bastante los FPS. Al dividir estas tareas entre varios hilos, se redujo el tiempo de procesamiento por frame, se subió el FPS y se mejoró la fluidez. El poder usar OpenMP facilitó pasar de código secuencial a paralelo gracias a su capacidad para manejar tareas en memoria compartida. Así, se logró mejorar el rendimiento de forma iterativa como lo reflejan los resultados.

El reto principal fue sincronizar bien los hilos y manejar los recursos compartidos. Se usaron directivas de sincronización, como secciones críticas y barreras, para evitar problemas de concurrencia y asegurar la integridad del programa. Aunque esto añadió algo de sobrecarga, el impacto en el rendimiento fue mínimo comparado con las mejoras en FPS que tuvo el programa paralelizado con respecto al secuencial teniendo un promedio de 61.10% de mejora en las distintas corridas que tuvo el programa con distintas cantidades de círculos.

En cuanto a la escalabilidad, el programa se mantuvo bien hasta cierto punto, pero encontramos límites al subir mucho el número de círculos o hilos. También se tuvo que, al pasar cierto número de hilos, las mejoras se vuelven mínimas o negativas, por la gestión de recursos. Esto subraya la importancia de ajustar la paralelización según el contexto.

Conclusión:

Durante las pruebas realizadas, se observó que el código paralelo mostró un rendimiento mayor en comparación con la versión secuencial a medida que se incrementaba la carga de trabajo. Este comportamiento puede explicarse por la capacidad del código paralelo para distribuir tareas entre múltiples hilos de ejecución, lo que permite que las operaciones se realicen de manera más eficiente y en menos tiempo. En aplicaciones donde la carga de trabajo es considerable, la paralelización puede ofrecer mejoras notables en el rendimiento, aprovechando al máximo los recursos de hardware disponibles.

Sin embargo, es importante destacar que la eficiencia de la paralelización depende del tipo de tarea y la naturaleza del problema. En situaciones con menor carga de trabajo, el overhead asociado con la gestión de hilos podría contrarrestar los beneficios de la paralelización, haciendo que el código secuencial sea una mejor opción en esos casos.

La elección entre un enfoque secuencial o paralelo debe considerar el nivel de carga de trabajo y las capacidades del hardware disponible. Para aplicaciones intensivas, en este caso, una mayor cantidad de círculos renderizados, la paralelización es una herramienta valiosa para mejorar el rendimiento y la eficiencia general del sistema.

Anexo 1: Diagrama de flujo

Anexo 2: Catálogo de Funciones

1. `float distance(float x1, float y1, float x2, float y2)`

   - Entradas:
     - `x1` (float): Coordenada X del primer punto.
     - `y1` (float): Coordenada Y del primer punto.
     - `x2` (float): Coordenada X del segundo punto.
     - `y2` (float): Coordenada Y del segundo punto.
   - Salidas:
     - `dist` (float): Distancia calculada entre los dos puntos.
   - Descripción:

     Esta función calcula la distancia euclidiana entre dos puntos dados por sus coordenadas `(x1, y1)` y `(x2, y2)`.

2. `SDL_Color getRandomColor()`

- Entradas:
    - Ninguna.
- Salidas:
    - `color` (SDL_Color): Color generado aleatoriamente.
- Descripción:

  Esta función genera un color aleatorio utilizando valores RGB y retorna un objeto `SDL_Color`.

3. `void handleCollision(Circle &a, Circle &b)`

- Entradas:
    - `a` (Circle &): Referencia a la primera partícula.
    - `b` (Circle &): Referencia a la segunda partícula.
- Salidas:
    - Ninguna.
- Descripción:

  Esta función detecta y maneja la colisión entre dos partículas. Si colisionan, ajusta sus velocidades, radios y colores, y asegura que las partículas no se solapen.

4. `void applyForces(std::vector<Circle> &circles)`

- Entradas:
    - `circles` (std::vector<Circle> &): Referencia a un vector que contiene todas las partículas.
- Salidas:
    - Ninguna.
- Descripción:

  Esta función aplica fuerzas de atracción/repulsión entre cada par de partículas. Usa `#pragma omp parallel for` para paralelizar el cálculo y `#pragma omp atomic` para asegurar la integridad de los datos compartidos.

5. `void attractToCenter(std::vector<Circle> &circles)`

- Entradas:
    - `circles` (std::vector<Circle> &): Referencia a un vector que contiene todas las partículas.
- Salidas:
    - Ninguna.
- Descripción:

  Esta función aplica una fuerza de atracción hacia el centro de la pantalla para cada partícula, actualizando sus velocidades. El proceso es paralelizado usando `#pragma omp parallel for`.

6. `float randomFloat()`

- Entradas:
    - Ninguna.
- Salidas:
    - `value` (float): Número aleatorio generado entre 0 y 1.
- Descripción:

  Esta función genera y retorna un número aleatorio de punto flotante entre 0 y 1.

7. `int main(int argc, char *argv[])`

- Entradas:
    - `argc` (int): Número de argumentos de la línea de comandos.
    - `argv` (char*[]): Array de argumentos de la línea de comandos.
- Salidas:
    - `status` (int): Código de estado de la ejecución, 0 si el programa finaliza correctamente, 1 si ocurre un error.
- Descripción:

  Esta es la función principal del programa. Captura los argumentos, inicializa SDL, crea las partículas, y ejecuta el bucle principal donde se aplican las fuerzas, se actualizan las posiciones, se manejan colisiones y se renderiza el contenido en pantalla. Al final, limpia los recursos utilizados.

Referencias

- *z/OS 2.4.0*. (s. f.). https://www.ibm.com/docs/en/zos/2.4.0?topic=processing-pragma-omp-critical
- *XL C/C++ for Linux 16.1.0*. (s. f.). https://www.ibm.com/docs/es/xl-c-and-cpp-linux/16.1.0?topic=parallelization-pragma-omp-atomic
- *z/OS 2.4.0*. (s. f.-b). https://www.ibm.com/docs/en/zos/2.4.0?topic=processing-pragma-omp-section-pragma-omp-sections