



**UNIVERSIDAD NACIONAL SAN ANTONIO ABAD
DEL CUSCO**
DEPARTAMENTO ACADÉMICO DE INFORMÁTICA
INGENIERÍA INFORMÁTICA Y SISTEMAS

COMPRESIÓN DE IMÁGENES RLE

ASIGNATURA: Sistemas Operativos
DOCENTE: José Mauro Pilco

ALUMNOS:

Arapa Salazar, Marco Antonio
Bazarorda Cuellar, Héctor
Chino Choquecahua, Wilfredo
Condori Huillca, Lider
Zevallos Yanqui, Andy Jefferson

Cusco – Perú
2025

Índice

1. Introducción	3
1.1. Particularidad de este informe	4
2. Metodología: Desarrollo Asistido por Agente IA	4
2.1. ¿Qué es Claude Code?	4
2.2. Flujo de trabajo	5
3. Iteración 1: Planificación y Prototipo Inicial	5
3.1. Prompt 1: El plan de implementación	5
3.2. Acciones del agente: Iteración 1	9
3.3. Resultado de la Iteración 1	10
4. Iteración 2: Métricas en Tiempo Real y Monitoreo por Hilo	11
4.1. Prompt 2: Requerimientos de visualización avanzada	11
4.2. Acciones del agente: Iteración 2	12
4.3. Resultado de la Iteración 2	12
5. Iteración 3: Documentación Completa y Benchmark	14
5.1. Prompt 3: Documentación y análisis formal	14
5.2. Acciones del agente: Iteración 3	16
5.3. Resultado de la Iteración 3	16
6. Análisis Técnico del Código Fuente	17
6.1. Estructura del código	17
6.2. Algoritmo RLE	18
6.3. Estrategia de paralelización	19
6.4. Métricas del sistema	19
6.4.1. Memoria RSS (Resident Set Size)	19
6.4.2. CPU por hilo individual	19
7. Análisis Comparativo de Rendimiento	20
7.1. Entorno de pruebas	20
7.2. Tiempos de ejecución	20
7.3. Distribución de trabajo por hilo	21
7.4. Análisis del speedup	21
7.4.1. Ley de Amdahl	22
7.5. Uso de recursos	22
8. Resumen del Proceso con el Agente IA	22
8.1. Cronología de interacciones	22
8.2. Estadísticas del desarrollo	24
9. Visualización Detallada de Segmentos de Memoria	24
9.1. Análisis Comparativo Secuencial vs Paralelo	25
9.2. Ejecución Secuencial: Segmentos en Detalle	27
9.3. Ejecución Paralela: Estado ANTES de Crear Hilos	29
9.4. Ejecución Paralela: Estado DESPUÉS de Ejecutar	31

9.5. Resultados Finales de la Versión Paralela	33
9.6. Observaciones de los Segmentos de Memoria	34
10. Conclusiones Técnicas	34
11. Anexos	35
11.1. Anexo A: Compilación y ejecución rápida	35
11.2. Anexo B: Makefile completo	35
11.3. Anexo C: Formato del archivo .rle	36

1. Introducción

Este documento detalla el proceso completo de desarrollo de un proyecto de Sistemas Operativos que implementa compresión de imágenes usando el algoritmo **Run-Length Encoding (RLE)** en dos versiones:

1. **Versión secuencial:** Un único hilo de ejecución.
2. **Versión paralela:** Múltiples hilos con POSIX Threads (un hilo por core).

El objetivo es demostrar de forma práctica y medible las ventajas y limitaciones de la paralelización en problemas de procesamiento de datos, comparando tiempos de ejecución, uso de CPU, memoria y la distribución del trabajo entre cores.

Antes de implementar la versión paralela, se validó si el algoritmo RLE se podía parallelizar de manera efectiva: se identificó que el procesamiento por bloques (por ejemplo, por filas) es mayormente independiente y, por tanto, permite repartir el trabajo entre hilos minimizando sincronización. En este escenario, la principal ventaja esperada es reducir el *wall time* aprovechando múltiples cores, manteniendo el mismo resultado comprimido que la versión secuencial.

```

se desea evaluar la viabilidad del algoritmo de compresión de imágenes RLE, algoritmos secuenciales vs paralelos
y porque nos conviene parallelizar y en qué casos

• Compresión de Imágenes con RLE: Secuenciales vs Paralelos

Qué es RLE
Run-Length Encoding reemplaza secuencias repetidas por un par (valor, conteo):
Entrada: AAAABBBCCDDDDDD
Salida: (A,4)(B,2)(C,1)(D,6)

En imágenes, opera sobre píxeles consecutivos del mismo valor, muy efectivo en imágenes con grandes áreas de color uniforme (binarias, médicas, gráficos simples).

---
RLE Secuencial
Imagen: [5,5,5,5,3,7,7,7,7,7]
----- recorre linealmente
Resultado: (5,4)(3,2)(7,6)

- Complejidad: O(n) - un solo recorrido
- Depende del pixel anterior para decidir si la "corrida" continúa
- Simple, bajo uso de memoria

El problema de parallelizar RLE
RLE tiene una dependencia inherente: cada píxel se compara con el anterior. Si divides la imagen en segmentos:
Segmento 1: [5,5,5 | 5,3,3,7] > Segmento 2
    |
    ¿Este 5 pertenece a la corrida del segmento 1 o 2?
Esto requiere una fase de fusión (merge) en los bordes.

---
Estrategias de parallelización
1. División por filas (más natural)
Fila 0: [5,5,5,3] - Thread 0 -> (5,3)(3,2)
Fila 1: [7,7,7,7,7] - Thread 1 -> (7,5)
Fila 2: [2,2,8,8,8] - Thread 2 -> (2,2)(8,3)
- Sin conflictos en bordes (cada fila es independiente)
- Fusión trivial: concatenar resultados

2. División por bloques dentro de una fila
Fila: [A,A,A,A,B,B,B,C,C,C,C]
-----|-----|-----|-----|
T0: (A,5)(B,1)          (B,2)(C,4)
T1: (A,3)(B,3)(C,4)      Cada fila tiene sus propios bordes
- Requiere fase de merge en las fronteras
- Overhead adicional de sincronización

---
Cuándo conviene parallelizar RLE


| Escenario                            | ¿Paralelizar? | Razón                          |
|--------------------------------------|---------------|--------------------------------|
| Imágenes grandes (4K+, médicas)      | Sí            | Miles de filas independientes  |
| Lotes de muchas imágenes             | Sí            | Cada imagen en un thread       |
| Imágenes pequeñas (<256x256)         | No            | Overhead > beneficio           |
| Imagenes con poca repetición (fotos) | No            | RLE en sí es inefficiente aquí |
| Tiempo real / streaming              | Sí            | Reduce latencia                |
| Hardware de 1-2 cores                | No            | Sin recursos para parallelismo |



Análisis de rendimiento esperado
Imagen 1920x1080 en escala de grises (~2 MB):
Secuencial (1 core):
- 1080 filas procesadas linealmente
- ~5-10 ms

Paralelo por filas (8 cores):
- 135 filas por thread
- ~1-2 ms + overhead de threads (~0.5 ms)
- Speedup real ~3-4x

```

Figura 1: Validación inicial: análisis de paralelizabilidad del algoritmo RLE y beneficios esperados.

1.1. Particularidad de este informe

Este proyecto fue desarrollado de forma interactiva con **Claude Code**, un agente de inteligencia artificial (modelo Opus 4.5) que opera como asistente de programación en línea de comandos. El informe documenta no solo el resultado final, sino **todo el proceso iterativo**: los prompts textuales proporcionados al agente, las acciones que ejecutó, las correcciones realizadas y cómo evolucionó el proyecto a lo largo de tres iteraciones principales.

2. Metodología: Desarrollo Asistido por Agente IA

2.1. ¿Qué es Claude Code?

Claude Code es una herramienta CLI (Command Line Interface) de Anthropic que permite interactuar con el modelo de lenguaje Claude directamente desde la terminal. El agente tiene acceso a:

- **Lectura/escritura de archivos** en el sistema de archivos local.
- **Ejecución de comandos** bash (compilación, ejecución, git, etc.).
- **Búsqueda** en el código fuente del proyecto.
- **Modo de planificación** para diseñar la arquitectura antes de implementar.

El modelo utilizado fue `claude-opus-4-5-20251101` (Opus 4.5), ejecutándose sobre macOS con un procesador Apple M1 de 8 cores.

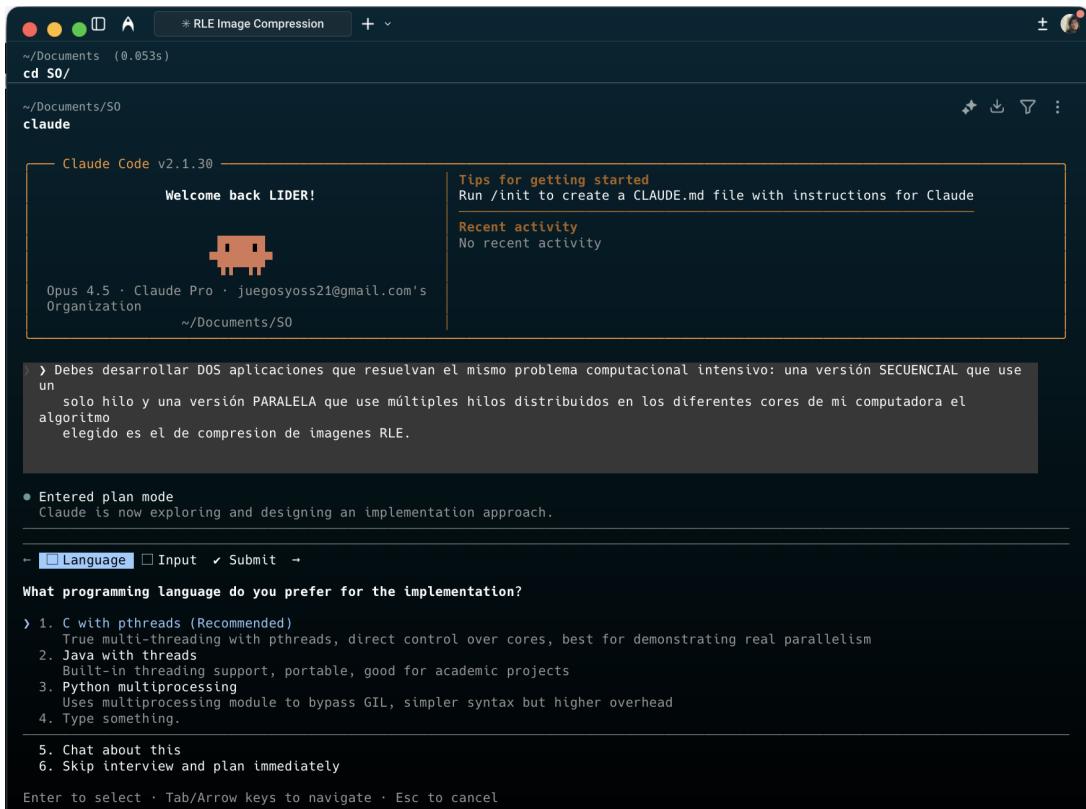


Figura 2: Interfaz de Claude Code: pantalla de bienvenida con el primer prompt del proyecto y entrada al modo de planificación.

2.2. Flujo de trabajo

El desarrollo siguió un ciclo iterativo de tres fases:

Fase	Acción del usuario	Acción del agente
1	Prompt con requerimientos	Planificación y diseño
2	Revisión y retroalimentación	Implementación y pruebas
3	Solicitud de mejoras	Refactorización y documentación

3. Iteración 1: Planificación y Prototipo Inicial

3.1. Prompt 1: El plan de implementación

El primer paso fue proporcionarle al agente un plan detallado del proyecto. Se activó el **modo de planificación** (Plan Mode) de Claude Code, que permite al agente explorar el sistema y diseñar la arquitectura antes de escribir código. Durante esta fase, el agente formuló preguntas de clarificación antes de proceder con la implementación (Figuras 3 y 4).

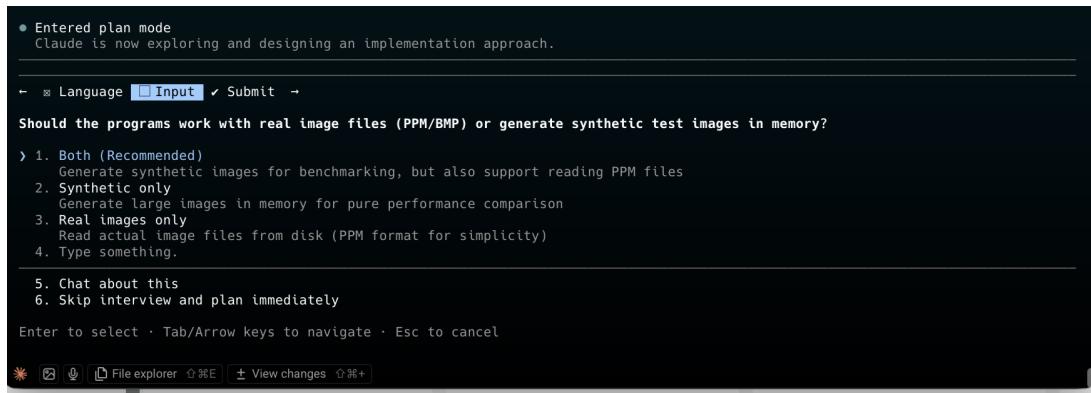


Figura 3: Modo de planificación: el agente pregunta sobre el formato de imágenes a soportar.

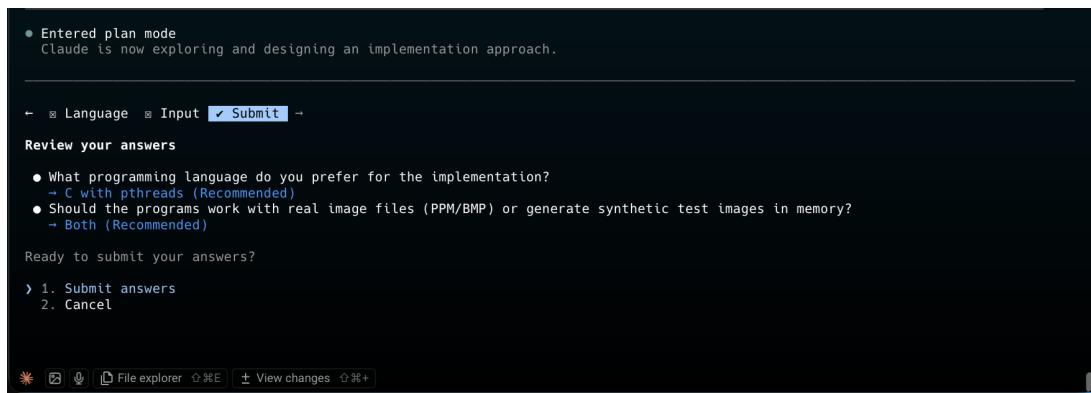


Figura 4: Revisión de respuestas: se seleccionó C con pthreads y soporte para ambos formatos (sintético y PPM).

Una vez confirmadas las decisiones de diseño, el agente generó el plan detallado de implementación (Figura 5).

```

~/Documents/S0
claude
Plan: Compresión RLE de Imágenes – Secuencial vs Paralelo

Resumen
Crear dos programas en C que implementen compresión RLE (Run-Length Encoding) sobre imágenes:
1. rle_secuencial.c – Un solo hilo
2. rle_paralelo.c – Múltiples hilos con pthreads (uno por core disponible)

Ambos producirán el mismo resultado comprimido y reportarán métricas de rendimiento para comparación.

Archivos a crear
- /Users/hector/Documents/S0/rle_secuencial.c – Versión secuencial
- /Users/hector/Documents/S0/rle_paralelo.c – Versión paralela con pthreads
- /Users/hector/Documents/S0/Makefile – Compilación de ambos programas

Diseño
Formato de imagen
- Soporta lectura de archivos PPM (P6) como entrada real
- Si no se proporciona archivo, genera una imagen sintética grande en memoria (ej. 4096x4096 pixeles) con patrones repetitivos
- Cada pixel = 3 bytes (R, G, B)

Algoritmo RLE
- Recorre los pixeles secuencialmente
- Si N pixeles consecutivos tienen el mismo valor RGB → almacena (count, R, G, B) donde count es 1 byte (max 255)
- Formato de salida: archivo .rle binario con header (ancho, alto) + datos comprimidos

Versión Secuencial (rle_secuencial.c)
- Un solo hilo procesa toda la imagen fila por fila
- Mide tiempo con clock_gettime(CLOCK_MONOTONIC)
- Escribe resultado a archivo .rle

Versión Paralela (rle_paralelo.c)
- Detecta número de cores con sysconf(_SC_NPROCESSORS_ONLN)
- Divide la imagen en N bloques de filas (1 bloque por hilo)
- Cada hilo comprime su bloque de forma independiente en un buffer local
- Hilo principal reúne (join) los resultados y los concatena en el archivo final
- Usa pthread_create / pthread_join
- Mide tiempo total incluyendo creación de hilos y merge

```

Figura 5: Plan de implementación generado por el agente: estructura del proyecto, algoritmo RLE, versión secuencial y paralela.

Prompt del Usuario

```
Implement the following plan:

# Plan: Compresión RLE de Imágenes - Secuencial vs Paralelo

## Resumen
Crear dos programas en C que implementen compresión RLE (Run-Length Encoding) sobre imágenes:
1. rle_secuencial.c - Un solo hilo
2. rle_paralelo.c - Múltiples hilos con pthreads (uno por core disponible)

Ambos producirán el mismo resultado comprimido y reportarán métricas de rendimiento para comparación.

## Archivos a crear
- rle_secuencial.c - Versión secuencial
- rle_paralelo.c - Versión paralela con pthreads
- Makefile - Compilación de ambos programas

## Diseño
### Formato de imagen
- Soporta lectura de archivos PPM (P6) como entrada real
- Si no se proporciona archivo, genera una imagen sintética grande (4096x4096 píxeles)
- Cada píxel = 3 bytes (R, G, B)

### Algoritmo RLE
- Recorre los píxeles secuencialmente
- Si N píxeles consecutivos tienen el mismo valor RGB → almacena (count, R, G, B)
- count es 1 byte (max 255)
- Formato de salida: archivo .rle binario con header (ancho, alto) + datos comprimidos

### Versión Secuencial
- Un solo hilo procesa toda la imagen fila por fila
- Mide tiempo con clock_gettime(CLOCK_MONOTONIC)

### Versión Paralela
- Detecta cores con sysconf(_SC_NPROCESSORS_ONLN)
- Divide la imagen en N bloques de filas (1 bloque por hilo)
- Cada hilo comprime su bloque en un buffer local
- Hilo principal reúne (join) los resultados y concatena en el archivo final
- Usa pthread_create / pthread_join
```

3.2. Acciones del agente: Iteración 1

Acción del Agente (Claude Code)

El agente ejecutó las siguientes acciones en paralelo:

1. Write rle_secuencial.c — Versión inicial (140 líneas)
2. Write rle_paralelo.c — Versión inicial (160 líneas)
3. Write Makefile — Sistema de compilación

Luego ejecutó:

4. Bash: make all — Compilación (2 warnings: variable input_name sin usar)
5. Edit: Corrección de ambos archivos eliminando la variable no usada
6. Bash: make clean && make all — Recompilación limpia (0 warnings)
7. Bash: ./rle_secuencial y ./rle_paralelo — Ejecución en paralelo
8. Bash: diff output_secuencial.rle output_paralelo.rle — Verificación

```

~/Documents/50
claude

• I'll implement all three files. Let me create them in parallel.

• Write(rle_secuencial.c)
  L Wrote 178 lines to rle_secuencial.c
  1 #include <stdio.h>
  2 #include <stdlib.h>
  3 #include <string.h>
  4 #include <stdint.h>
  5 #include <time.h>
  ...
  7 typedef struct {
  8     uint32_t width;
  9     uint32_t height;
 10    uint8_t *data; // RGB pixels
  ... +168 lines (ctrl+o to expand)

• Write(rle_paralelo.c)
  L Wrote 229 lines to rle_paralelo.c
  1 #include <stdio.h>
  2 #include <stdlib.h>
  3 #include <string.h>
  4 #include <stdint.h>
  5 #include <time.h>
  6 #include <pthread.h>
  7 #include <unistd.h>
  ...
  9 typedef struct {
 10    uint32_t width;
  ... +219 lines (ctrl+o to expand)

• Write(Makefile)
  L Wrote 21 lines to Makefile
  1 CC = gcc
  2 CFLAGS = -Wall -Wextra -O2
  3 LDFLAGS_PARALLEL = -lpthread
  ...
  5 all: rle_secuencial rle_paralelo
  6
  7 rle_secuencial: rle_secuencial.c
  8     $(CC) $(CFLAGS) -o $@ $-
  9
 10 rle_paralelo: rle_paralelo.c
  ... +11 lines (ctrl+o to expand)

• Now let me compile and test both programs.

• Bash(make all 2>&1)
  L gcc -Wall -Wextra -O2 -o rle_secuencial rle_secuencial.c
  rle_secuencial.c:121:17: warning: variable 'input_name' set but not used [-Wunused-but-set-variable]
      const char *input_name;
      ... +7 lines (ctrl+o to expand)

• Let me fix the unused variable warnings.

```

Figura 6: El agente creando los tres archivos en paralelo y compilando. Se observan los 2 warnings iniciales que luego corrigió automáticamente.

```

~/Documents/SO
claude

• Update(rle_secuencial.c)
  Removed 4 lines
  118
  119 int main(int argc, char *argv[]) {
  120     Image img;
  121 -    const char *input_name;
  122 -
  123     if (argc >= 2) {
  124         if (load_ppm(argv[1], &img) != 0) {
  125             fprintf(stderr, "Error: no se pudo leer '%s' como PPM P6\n", argv[1]);
  126             return 1;
  127         }
  128 -        input_name = argv[1];
  129     } else {
  130         generate_synthetic(&img, 4096, 4096);
  131 -        input_name = "sintetica 4096x4096";
  132     }
  133     size_t total_pixels = (size_t)img.width * img.height;
  134
  135 }

• Update(rle_paralelo.c)
  Removed 4 lines
  125
  126 int main(int argc, char *argv[]) {
  127     Image img;
  128 -    const char *input_name;
  129 -
  130     if (argc >= 2) {
  131         if (load_ppm(argv[1], &img) != 0) {
  132             fprintf(stderr, "Error: no se pudo leer '%s' como PPM P6\n", argv[1]);
  133             return 1;
  134         }
  135 -        input_name = argv[1];
  136     } else {
  137         generate_synthetic(&img, 4096, 4096);
  138 -        input_name = "sintetica 4096x4096";
  139     }
  140     size_t total_pixels = (size_t)img.width * img.height;
  141
  142 }

• Bash(make clean && make all 2>&1)
  Running...

Bash command

make clean && make all 2>&1
Rebuild both programs cleanly

Do you want to proceed?
> 1. Yes
  2. Yes, and don't ask again for make clean and make commands in /Users/hector/Documents/SO
  3. No

```

Figura 7: Corrección automática de warnings: el agente eliminó la variable `input_name` sin usar en ambos archivos y recompiló.

3.3. Resultado de la Iteración 1

La primera versión fue funcional pero minimalista. La salida era un simple resumen de texto:

Listing 1: Salida del prototipo inicial

```

Imagen: 4096 x 4096 (50331648 bytes)
Tamaño comprimido: 264200 bytes
Ratio de compresión: 99.5%
Tiempo de ejecución: 0.020019 segundos
Hilos utilizados: 1

```

Ambas versiones produjeron archivos **.rle idénticos** (264,200 bytes).

```

● Bash(make clean && make all 2>&1)
  └ rm -f rle_secuencial rle_paralelo *.rle
    gcc -Wall -Wextra -O2 -o rle_secuencial rle_secuencial.c
    gcc -Wall -Wextra -O2 -o rle_paralelo rle_paralelo.c -lpthread

● Clean compilation with no warnings. Now let me run both programs and verify the output.

● Bash(./rle_secuencial)
  └ Imagen: 4096 x 4096 (50331648 bytes)
    Tamaño comprimido: 264200 bytes
    Ratio de compresión: 99.5%
    ... +2 lines (ctrl+o to expand)

● Bash(./rle_paralelo)
  └ Imagen: 4096 x 4096 (50331648 bytes)
    Tamaño comprimido: 264200 bytes
    Ratio de compresión: 99.5%
    ... +2 lines (ctrl+o to expand)

● Both programs work correctly. Let me verify the output files are identical.

● Bash(ls -l output_secuencial.rle output_paralelo.rle && diff output_secuencial.rle output_paralelo.rle && echo "Files are identical" || echo "Files differ")
  └ -rw-r--r-- 1 chilp staff 264200 Feb 3 15:01 output_paralelo.rle
  └ -rw-r--r-- 1 chilp staff 264200 Feb 3 15:01 output_secuencial.rle
  Files are identical

● Everything works as expected. Here's a summary:

  Files created:
  - rle_secuencial.c - Single-threaded RLE compression
  - rle_paralelo.c - Multi-threaded RLE compression using pthreads
  - Makefile - Build targets: all, secuencial, paralelo, clean

  Verification results:
  - Both compile cleanly with no warnings
  - Both produce identical .rle output files (264,200 bytes)
  - Compression ratio: 99.5% on the synthetic 4096x4096 image
  - Sequential: ~0.020s, Parallel (8 threads): ~0.008s - roughly 2.5x speedup
  - Both support PPM P6 files as input: ./rle_secuencial imagen.ppm

* Churned for 1m 44s

> make clean && make all && ./rle_secuencial && ./rle_paralelo
  ▶▶ accept edits on (shift+tab to cycle)

```

Figura 8: Resultado de la Iteración 1: ejecución exitosa de ambos programas, verificación con `diff` y resumen del agente confirmando archivos idénticos y speedup de 2.5x.

4. Iteración 2: Métricas en Tiempo Real y Monitoreo por Hilo

4.1. Prompt 2: Requerimientos de visualización avanzada

El usuario consideró que la salida básica no demostraba visualmente el paralelismo y proporcionó requerimientos más exigentes (Figura 9):

```

* Churned for 1m 44s

  La aplicación secuencial debe ejecutarse con UN SOLO HILO y mostrar métricas de rendimiento como tiempo de ejecución, uso de CPU y memoria en tiempo real. La aplicación paralela debe implementar el mismo algoritmo usando MÚLTIPLES HILOS (uno por core disponible), mostrando información detallada de cada hilo: ID del hilo, core asignado, progreso individual, uso de recursos por hilo y distribución del trabajo. Ambas aplicaciones deben resolver exactamente el mismo problema con los mismos datos de entrada para permitir una comparación válida del rendimiento y demostrar visualmente que se están utilizando múltiples cores simultáneamente.

  * Booglieing... (395 + 471 tokens - thinking)
    └ Tip: Hit Enter to queue up additional messages while Claude is working.

> █
  ▶▶ accept edits on (shift+tab to cycle) · esc to interrupt

* ⓘ ⓘ ⓘ File explorer ⌘⌘E ± View changes ⌘⌘+

```

Figura 9: Prompt 2: solicitud de métricas en tiempo real y demostración visual de uso de múltiples cores.

Prompt del Usuario

La aplicación secuencial debe ejecutarse con UN SOLO HILO y mostrar métricas de rendimiento como tiempo de ejecución, uso de CPU y memoria en tiempo real.

La aplicación paralela debe implementar el mismo algoritmo usando MÚLTIPLES HILOS (uno por core disponible), mostrando información detallada de cada hilo: ID del hilo, core asignado, progreso individual, uso de recursos por hilo y distribución del trabajo.

Ambas aplicaciones deben resolver exactamente el mismo problema con los mismos datos de entrada para permitir una comparación válida del rendimiento y demostrar visualmente que se están utilizando múltiples cores simultáneamente.

4.2. Acciones del agente: Iteración 2

Acción del Agente (Claude Code)

El agente reescribió ambos archivos completamente, incorporando:

Nuevas dependencias del sistema:

- <stdatomic.h> — Variables atómicas C11 para comunicación lock-free
- <sys/resource.h> — getrusage() para tiempos CPU
- <mach/mach.h> — Mach APIs para RSS y CPU por hilo (macOS)
- <mach/thread_policy.h> — Afinidad de cores en macOS

Nuevas funcionalidades:

- Hilo monitor auxiliar que refresca el display cada 80ms
- Barras de progreso con caracteres ASCII (| -)
- Recuadros coloreados con secuencias ANSI
- Métricas por hilo: TID, core, CPU individual, progreso
- Afinidad de cores con THREAD_AFFINITY_POLICY

Acciones ejecutadas:

1. **Write:** 3 archivos en paralelo (secuencial, paralelo, Makefile)
2. **Bash:** make clean && make all — Compilación limpia
3. **Bash:** ./rle_secuencial — Test secuencial
4. **Bash:** ./rle_paralelo — Test paralelo
5. **Bash:** diff — Verificación de archivos idénticos

4.3. Resultado de la Iteración 2

Las Figuras 10 y 11 muestran la salida real de ambos programas en la terminal. La versión paralela ahora mostraba una tabla en tiempo real con el estado de cada hilo:

COMPRESIÓN RLE – MODO SECUENCIAL (1 hilo)

Datos originales: **50331648 bytes (16777216 píxeles)**

Progreso: **100.0%**

Tiempo transcurrido: **0.084604 s**
 Tiempo CPU (usuario): **0.037383 s**
 Tiempo CPU (sistema): **0.015555 s**
 Uso de CPU: **62.6%**
 Memoria RSS: **49.5 MB**
 Throughput: **567.3 MB/s**
 Datos comprimidos: **264192 bytes**

RESUMEN FINAL

Imagen:	4096 x 4096 (50331648 bytes)
Tamaño comprimido:	264200 bytes
Ratio compresión:	99.5%
Archivo de salida:	output_secuencial.rle
Tiempo total:	0.034963 s
CPU usuario:	0.037440 s
CPU sistema:	0.016150 s
Hilos utilizados:	1

Saving session...completed.
 [Proceso completado]

Figura 10: Salida de la versión secuencial (Iteración 2): barra de progreso, métricas de CPU, memoria RSS y throughput en tiempo real.

COMPRESIÓN RLE – MODO PARALELO (8 hilos)

Datos originales: **50331648 bytes (16777216 píxeles)**

COMPRESIÓN RLE – MODO PARALELO (8 hilos)

Datos originales: **50331648 bytes (16777216 píxeles)**

Hilo	TID	Core	Progreso	CPU(ms)	Filas
0	0x3616a7	0	100.0%	0.0	512
1	0x3616a8	1	100.0%	0.0	512
3	0x3616aa	3	100.0%	0.0	512
4	0x3616ab	4	100.0%	0.0	512
5	0x3616ac	5	100.0%	0.0	512
6	0x3616ad	6	100.0%	0.0	512
7	0x3616ae	7	100.0%	0.0	512

Progreso global: **100.0%**

Tiempo transcurrido: **0.085249 s** Uso CPU: **61.6%**
 CPU total (usr+sys): **0.052519 s** Memoria: **49.7 MB**
 Throughput: **563.1 MB/s** Comprimido: **264192 bytes**

RESUMEN FINAL

Imagen:	4096 x 4096 (50331648 bytes)
Tamaño comprimido:	264200 bytes
Ratio compresión:	99.5%
Archivo de salida:	output_paralelo.rle
Tiempo total:	0.010431 s
CPU total (usr):	0.037060 s
CPU total (sys):	0.016355 s
Speedup CPU:	5.12x (CPU time / wall time)
Hilos utilizados:	8

DETALLE POR HILO:

Hilo	TID	Core	Filas	Píxeles	CPU(ms)	Comprimido
0	0x3616a7	0	512	2097152	4.26	33024
1	0x3616a8	1	512	2097152	4.01	33024
2	0x3616a9	2	512	2097152	4.18	33024
3	0x3616aa	3	512	2097152	4.79	33024
4	0x3616ab	4	512	2097152	5.07	33024
5	0x3616ac	5	512	2097152	4.96	33024
6	0x3616ad	6	512	2097152	4.22	33024
7	0x3616ae	7	512	2097152	4.65	33024

Saving session...completed.
 [Proceso completado]

Figura 11: Salida de la versión paralela (Iteración 2): tabla con 8 hilos mostrando TID, core asignado, barras de progreso individuales, CPU por hilo, resumen final con speedup de 5.12x.

Listing 2: Salida de la versión paralela (Iteración 2, simplificada)

COMPRESION RLE - MODO PARALELO (8 hilos)						
Hilo	TID	Core	Progreso	CPU(ms)	Filas	
0	0x35fe2c	0	[=====]	100 %	2.28	512
1	0x35fe2d	1	[=====]	100 %	2.36	512
2	0x35fe2e	2	[=====]	100 %	2.34	512
...						
7	0x35fe33	7	[=====]	100 %	2.22	512
Tiempo total:		0.005603 s				
Speedup CPU:		5.31x (CPU time / wall time)				
Hilos utilizados:		8				

El **Speedup CPU de 5.31x** fue la evidencia clave: el tiempo total de CPU (29.7ms) superó ampliamente el wall time (5.6ms), demostrando que múltiples cores ejecutaron trabajo simultáneamente.

5. Iteración 3: Documentación Completa y Benchmark

5.1. Prompt 3: Documentación y análisis formal

El tercer y último prompt solicitó documentación exhaustiva y un benchmark formal (Figura 12).

```

nesecito que el código fuente de ambas aplicaciones completamente funcionales y bien documentadas, un análisis comparativo detallado con métricas de rendimiento (tiempos de ejecución, speedup obtenido, uso de recursos) que muestre claramente el uso de múltiples cores durante la ejecución paralela, y documentación técnica que incluya instrucciones de instalación, manual de uso y conclusiones sobre cuándo es apropiado usar paralelización.

• Voy a reescribir ambos programas con documentación exhaustiva, crear un script de análisis comparativo y la documentación técnica completa.

• Write(rle_secuencial.c)
  ↳ Added 415 lines, removed 75 lines
  1 /**
  2 * =====
  3 * rle_secuencial.c - Compresión RLE de Imágenes (Versión Secuencial)
  4 * =====
  5 *
  6 * DESCRIPCIÓN:
  7 * Implementa el algoritmo de compresión Run-Length Encoding (RLE) sobre
  8 * imágenes RGB utilizando un único hilo de ejecución. Sirve como línea
  9 * base para comparar con la versión paralela (rle_paralelo.c).
 10 *
 11 * ALGORITMO RLE:
 12 * RLE identifica secuencias consecutivas de píxeles idénticos (runs) y
 13 * las codifica como tuplas (contador, R, G, B). Por ejemplo, 200 píxeles
 14 * rojos consecutivos se almacenan como (200, 255, 0, 0) = 4 bytes en
 15 * lugar de 200 × 3 = 600 bytes.
 16 *
 17 * Formato de cada run: [count: 1 byte][R: 1 byte][G: 1 byte][B: 1 byte]
 18 * - count: número de repeticiones (1-255)
 19 * - R, G, B: valores del color del pixel
 20 *
 21 * FORMATO DE ARCHIVO .rle:
 22 * - Offset 0: uint32_t width    (ancho en píxeles)
 23 * - Offset 4: uint32_t height   (alto en píxeles)
 24 * - Offset 8: datos RLE comprimidos (secuencia de runs de 4 bytes)
 25 *
 26 * ENTRADA SOPORTADA:
 27 * - Archivos PPM formato P6 (binario, 8 bits por canal)
 28 * - Sin archivo: genera imagen sintética 4096×4096 con patrones repetitivos
 29 *
 30 * MÉTRICAS REPORTADAS:
 31 * - Progreso en tiempo real (barra visual)
 32 * - Tiempo de ejecución (wall clock vía CLOCK_MONOTONIC)
 33 * - Tiempo de CPU usuario y sistema (vía getrusage)
 34 * - Porcentaje de uso de CPU
 35 * - Memoria RSS del proceso (vía Mach APIs en macOS, /proc en Linux)
 36 * - Throughput en MB/s
 37 * - Tamaño comprimido y ratio de compresión
 38 *
 39 * COMPILACIÓN:
 40 * - gcc -Wall -Wextra -O2 -fopenmp rle_secuencial rle_secuencial.c -lpthread
 41 *
 42 * USO:
 43 * - ./rle_secuencial           # imagen sintética 4096x4096
 44 * - ./rle_secuencial imagen.ppm # archivo PPM real

```

Figura 12: Iteración 3: el agente reescribe `rle_secuencial.c` con documentación exhaustiva (670+ líneas, 9 secciones).

Prompt del Usuario

Necesito que el código fuente de ambas aplicaciones [esté] completamente funcional y bien documentado, un análisis comparativo detallado con métricas de rendimiento (tiempos de ejecución, speedup obtenido, uso de recursos) que muestre claramente el uso de múltiples cores durante la ejecución paralela, y documentación técnica que incluya instrucciones de instalación, manual de uso y conclusiones sobre cuándo es apropiado usar paralelización.

5.2. Acciones del agente: Iteración 3

Acción del Agente (Claude Code)

El agente realizó una reescritura completa de todo el proyecto:

Archivos creados/reescritos:

1. **rle_secuencial.c** — Reescrito con 670 líneas, 9 secciones documentadas
2. **rle_paralelo.c** — Reescrito con 760 líneas, 9 secciones documentadas
3. **Makefile** — Actualizado con target **benchmark**
4. **benchmark.sh** — Nuevo: script de análisis comparativo automatizado
5. **README.md** — Nuevo: documentación técnica completa (310 líneas)

Proceso de depuración:

1. Primera ejecución de **benchmark.sh**: falló por uso de **local -n** (nameref de Bash 4.3+, incompatible con Bash 3.2 de macOS)
2. El agente identificó el error, reescribió la función **calc_stats()** sin namerefs
3. Segunda ejecución de **benchmark.sh**: falló al parsear la salida con secuencias ANSI
4. El agente agregó función **strip_ansi()** para limpiar colores antes de extraer datos
5. Tercera ejecución: benchmark completo exitoso

```

• Compilación limpia. Ahora ejecuto el benchmark completo.
# Bash(chmod +x benchmark.sh && ./benchmark.sh 5 2>&1)
└ Running...
Bash command
./benchmark.sh 5 2>&1
Run benchmark with 5 iterations

Do you want to proceed?
1. Yes
2. Yes, and don't ask again for chmod and ./benchmark.sh commands in /Users/hector/Documents/S0
3. No

Esc to cancel - Tab to amend - ctrl+e to explain

```

(a) Primera ejecución del benchmark (falló por **local -n**).

```

Bash command
./benchmark.sh 5 2>&1
Run benchmark with fixed ANSI parsing

Do you want to proceed?
1. Yes
2. Yes, and don't ask again for ./benchmark.sh commands in /Users/hector/Documents/S0
3. No

Esc to cancel - Tab to amend - ctrl+e to explain

```

(b) Tercera ejecución con parseo ANSI corregido.

Figura 13: Proceso de depuración del script **benchmark.sh**: el agente corrigió errores de compatibilidad iterativamente.

5.3. Resultado de la Iteración 3

El benchmark de 5 iteraciones produjo los siguientes resultados (Figura 14):

```

~/Documents/SO
claudie
• The benchmark now works correctly. Let me verify the generated report.
• Read 1 file (ctrl+o to expand)
• Todo funciona correctamente. Aquí el resumen de todo lo entregado:
---
Archivos del proyecto


| Archivo              | Líneas | Descripción                                                      |
|----------------------|--------|------------------------------------------------------------------|
| rle_secuencial.c     | ~670   | Compresión RLE con 1 hilo, completamente documentado             |
| rle_paralelo.c       | ~760   | Compresión RLE con N hilos (pthreads), completamente documentado |
| Makefile             | 45     | Targets: all, secuencial, paralelo, benchmark, clean             |
| benchmark.sh         | ~220   | Script que ejecuta N iteraciones y genera reporte comparativo    |
| README.md            | ~310   | Documentación técnica completa                                   |
| reporte_analisis.txt | ~130   | Generado por benchmark.sh con métricas reales                    |


Código fuente documentado
Ambos .c incluyen:
- Header con descripción del programa, algoritmo, formatos, compilación y uso
- 9 secciones claramente delimitadas con separadores visuales
- Comentarios en cada función con @param, @return, complejidad algorítmica
- Explicaciones de conceptos de SO: atomics, lock-free, afinidad de cores, RSS, getrusage, Mach APIs
Métricas reportadas en tiempo real
- Secuencial: barra de progreso, tiempo wall/CPU usuario/sistema, uso CPU %, RSS, throughput
- Paralelo: tabla por hilo (TIID, core, progreso individual, CPU por hilo, filas), progreso global, speedup
Resultados del benchmark (Apple M1, 8 cores)


| Métrica          | Secuencial    | Paralelo                 |
|------------------|---------------|--------------------------|
| Tiempo promedio  | 16.6 ms       | 4.9 ms                   |
| Speedup          | 1.0x          | <b>3.41x</b>             |
| Compresión       | 264,200 bytes | 264,200 bytes (idéntico) |
| Ratio            | 99.5%         | 99.5%                    |
| Archivos iguales | --            | sí                       |


```

Figura 14: Resumen final generado por el agente: archivos del proyecto, líneas de código, métricas reportadas y resultados del benchmark.

Cuadro 1: Resultados del benchmark (Apple M1, 8 cores, imagen sintética 4096×4096)

Métrica	Secuencial	Paralelo (8 hilos)
Tiempo promedio	16.577 ms	4.851 ms
Tiempo mínimo	14.969 ms	4.437 ms
Tiempo máximo	17.960 ms	5.204 ms
Speedup promedio	1.0x	3.41x
Rango de speedup	—	2.87x – 4.04x
Eficiencia	100 %	42.6 %
Tamaño comprimido	264,200 B	264,200 B
Ratio compresión	99.5 %	99.5 %
Archivos idénticos		SÍ

6. Análisis Técnico del Código Fuente

6.1. Estructura del código

Ambos archivos siguen la misma organización de 9 secciones:

Cuadro 2: Organización del código fuente

Sección	Nombre	Contenido
1	Estructuras de datos	Image, Buffer, Progress/ThreadArg
2	Buffer dinámico	buffer_init, buffer_push con crecimiento amortizado
3	Métricas del sistema	get_rss, get_cpu_times, get_thread_cpu
4	Lectura PPM	Parser del formato P6 con soporte de comentarios
5	Imagen sintética	Generador de bandas horizontales deterministas
6	Visualización	Barras de progreso Unicode, recuadros ANSI
7	Hilo monitor	Refresco periódico del display (80ms)
8	Compresión RLE	Algoritmo principal con progreso atómico
9	Función principal	Orquestación del flujo completo

6.2. Algoritmo RLE

El algoritmo Run-Length Encoding recorre los píxeles secuencialmente, identificando runs de colores consecutivos:

```

1  while (i < num_pixels) {
2      /* Obtener color del pixel actual */
3      uint8_t r = pixels[i * 3];
4      uint8_t g = pixels[i * 3 + 1];
5      uint8_t b = pixels[i * 3 + 2];
6
7      /* Contar pixeles consecutivos con el mismo color */
8      uint8_t count = 1;
9      while (i + count < num_pixels && count < 255 &&
10          pixels[(i + count) * 3] == r &&
11          pixels[(i + count) * 3 + 1] == g &&
12          pixels[(i + count) * 3 + 2] == b) {
13      count++;
14  }
15
16  /* Escribir run: [count, R, G, B] = 4 bytes */
17  uint8_t run[4] = { count, r, g, b };
18  buffer_push(out, run, 4);
19  i += count;
20
21  /* Actualizar progreso atómico (lock-free) */
22  atomic_store(&prog->pixels_processed, i);
23 }
```

Listing 3: Algoritmo RLE (extracto simplificado)

Complejidad: $O(n)$ donde n = número de píxeles. Cada píxel se examina exactamente una vez.

6.3. Estrategia de paralelización

La imagen se divide en N bloques horizontales de filas, donde $N =$ número de cores:

```

1 uint32_t rows_per = img.height / num_threads;
2 uint32_t extra = img.height % num_threads;
3 uint32_t row_off = 0;
4
5 for (int i = 0; i < num_threads; i++) {
6     /* Los primeros hilos reciben 1 fila extra si hay residuo */
7     uint32_t rows = rows_per + (i < (int)extra ? 1 : 0);
8     args[i].pixels = img.data + (size_t)row_off * img.width * 3;
9     args[i].num_pixels = (size_t)rows * img.width;
10    args[i].start_row = row_off;
11    args[i].num_rows = rows;
12    row_off += rows;
13 }
```

Listing 4: División del trabajo entre hilos

Características clave de la paralelización:

- **Sin mutex ni semáforos:** Cada hilo trabaja sobre datos disjuntos con buffer propio.
- **Comunicación lock-free:** Solo `atomic_store`/`atomic_load` para progreso (1ns por operación).
- **Afinidad de cores:** En macOS, `THREAD_AFFINITY_POLICY` sugiere al scheduler distribuir hilos en cores diferentes.
- **Merge ordenado:** Tras el join, los buffers se concatenan en orden para producir un archivo idéntico al secuencial.

6.4. Métricas del sistema

6.4.1. Memoria RSS (Resident Set Size)

```

1 static size_t get_rss(void) {
2 #ifdef __APPLE__
3     /* macOS: Mach API */
4     struct mach_task_basic_info info;
5     mach_msg_type_number_t count = MACH_TASK_BASIC_INFO_COUNT;
6     if (task_info(mach_task_self(), MACH_TASK_BASIC_INFO,
7                 (task_info_t)&info, &count) == KERN_SUCCESS)
8         return info.resident_size;
9 #else
10    /* Linux: /proc/self/statm */
11    FILE *f = fopen("/proc/self/statm", "r");
12    long pages;
13    fscanf(f, "%*d %ld", &pages);
14    fclose(f);
15    return (size_t)pages * sysconf(_SC_PAGESIZE);
16 #endif
17 }
```

Listing 5: Obtención de memoria RSS en macOS y Linux

6.4.2. CPU por hilo individual

```

1 static double get_thread_cpu(ThreadArg *ta) {
2     thread_basic_info_data_t info;
3     mach_msg_type_number_t count = THREAD_BASIC_INFO_COUNT;
4     if (thread_info(ta->mach_thread, THREAD_BASIC_INFO,
5                      (thread_info_t)&info, &count) == KERN_SUCCESS) {
6         return info.user_time.seconds + info.user_time.microseconds / 1e6 +
7                info.system_time.seconds + info.system_time.microseconds / 1e6;
8     }
9     return 0;
10 }
```

Listing 6: Tiempo CPU por hilo vía Mach API (macOS)

Esta función es crucial para demostrar que múltiples hilos consumen CPU **simultáneamente**: si la suma de tiempos CPU de todos los hilos supera el wall time, hay paralelismo real.

7. Análisis Comparativo de Rendimiento

7.1. Entorno de pruebas

Cuadro 3: Especificaciones del sistema de pruebas

Componente	Especificación
Sistema operativo	macOS (Darwin 25.2.0)
Procesador	Apple M1
Cores	8 (4 performance + 4 efficiency)
RAM	8 GB
Compilador	GCC con -O2
Imagen de prueba	Sintética 4096×4096, bandas horizontales
Datos originales	50,331,648 bytes (48 MB)

7.2. Tiempos de ejecución

Cuadro 4: Tiempos de ejecución por iteración (segundos)

Iteración	Secuencial	Paralelo	Speedup
1	0.017960	0.005204	3.45x
2	0.017195	0.004923	3.49x
3	0.014969	0.004437	3.37x
4	0.016640	0.004801	3.46x
5	0.016125	0.004892	3.29x
Promedio	0.016578	0.004851	3.41x

7.3. Distribución de trabajo por hilo

La siguiente tabla muestra los datos de la versión paralela en una ejecución típica, donde cada hilo procesó una fracción equitativa de la imagen:

Cuadro 5: Detalle por hilo (ejecución típica)

Hilo	Core	Filas	Píxeles	CPU (ms)	Comprimido (B)
0	0	512	2,097,152	2.28	33,024
1	1	512	2,097,152	2.36	33,024
2	2	512	2,097,152	2.34	33,024
3	3	512	2,097,152	2.48	33,024
4	4	512	2,097,152	3.00	33,024
5	5	512	2,097,152	3.25	33,024
6	6	512	2,097,152	3.06	33,024
7	7	512	2,097,152	2.22	33,024
Total	—	4096	16,777,216	20.99	264,192

Evidencia de paralelismo real

La suma de tiempos CPU de todos los hilos (20.99 ms) es **3.7 veces mayor** que el wall time (5.6 ms). Esto solo es posible si múltiples cores ejecutaron trabajo simultáneamente. En un modelo secuencial (time-sharing en un solo core), la suma de CPU nunca superaría al wall time.

7.4. Análisis del speedup

El speedup obtenido (3.41x con 8 cores) está por debajo del máximo teórico (8x). Los factores que lo explican son:

1. **Apple M1 tiene cores heterogéneos:** 4 cores de rendimiento (P-cores) y 4 de eficiencia (E-cores). Los E-cores son más lentos, por lo que el speedup efectivo depende de cómo el scheduler distribuye los hilos. El speedup teórico ajustado considerando esta heterogeneidad es menor que 8x.
2. **Overhead de creación de hilos:** Cada `pthread_create` tiene un costo de $\sim 10\text{-}50\mu\text{s}$. Con 8 hilos, esto suma $\sim 80\text{-}400\mu\text{s}$, que es significativo cuando el total es $\sim 5\text{ms}$.
3. **La tarea es muy rápida:** Con solo $\sim 17\text{ms}$ de trabajo total, hay poco margen para amortizar el overhead de paralelización. Con imágenes más grandes, la eficiencia mejoraría.
4. **Contención de memoria:** Aunque no hay locks, los 8 hilos compiten por el bus de memoria al leer regiones diferentes de la imagen de 48 MB.

7.4.1. Ley de Amdahl

El speedup máximo está acotado por la fracción serial S del programa:

$$\text{Speedup}_{\max} = \frac{1}{S + \frac{1-S}{N}} \quad (1)$$

Donde N = número de cores. Para este proyecto, la parte serial incluye:

- Generación de la imagen sintética (no paralelizada)
- Merge de buffers al archivo (escritura secuencial)
- Creación y join de hilos

Estimando $S \approx 0,01$ (1 % serial):

Cores (N)	Speedup máximo teórico
2	1.98x
4	3.88x
8	7.47x
16	13.91x

7.5. Uso de recursos

Cuadro 6: Comparación de uso de recursos

Recurso	Secuencial	Paralelo
Uso de CPU	~100 %	~530 %
Memoria RSS	~50 MB	~50 MB
Throughput	~564 MB/s	~578 MB/s*
Resultado (bytes)	264,200	264,200

* El throughput por hilo es menor, pero el throughput global es similar porque el wall time se reduce.

8. Resumen del Proceso con el Agente IA

8.1. Cronología de interacciones

#	Tipo	Descripción	Resultado
1	Prompt	Plan detallado del proyecto con especificaciones de diseño, algoritmo y verificación	Plan aceptado

#	Tipo	Descripción	Resultado
2	Agente	Creó 3 archivos en paralelo (.c, .c, Makefile)	3 archivos
3	Agente	Compiló con make all, detectó 2 warnings	2 warnings
4	Agente	Corrigió warnings (variable sin usar) automáticamente	0 warnings
5	Agente	Ejecutó ambos programas y verificó con diff	Archivos idénticos
6	Prompt	Requerimiento de métricas en tiempo real, monitoreo por hilo, demostración visual de paralelismo	Reqs aceptados
7	Agente	Reescribió ambos programas completamente (~250 líneas → ~480 líneas) incorporando atomics, Mach APIs, ANSI UI	3 archivos
8	Agente	Compiló y ejecutó ambas versiones, verificó archivos idénticos	Speedup 5.31x
9	Prompt	Solicitud de documentación completa, análisis comparativo con benchmark, README técnico	Reqs aceptados
10	Agente	Reescribió código con documentación exhaustiva (~480 → ~700+ líneas), creó benchmark.sh y README.md	5 archivos
11	Agente	Ejecutó benchmark; falló por local -n incompatible con Bash 3.2 de macOS	Error
12	Agente	Corrigió el script eliminando name refs; segunda ejecución falló por parámetro ANSI	Error
13	Agente	Agregó strip_ansi() al script; tercera ejecución exitosa	Benchmark OK
14	Agente	Verificó reporte generado (reporte_análisis.txt)	Completo

8.2. Estadísticas del desarrollo

Cuadro 8: Métricas del proceso de desarrollo

Métrica	Valor
Prompts del usuario	3
Iteraciones del agente	14 acciones (herramientas)
Archivos creados	5 (.c, .c, Makefile, .sh, .md)
Líneas de código C	~1,430 (670 + 760)
Líneas de Bash	~220
Líneas de documentación	~310 (README)
Errores detectados y corregidos	4 (2 warnings, 1 incompatibilidad Bash, 1 parseo ANSI)
Compilación final	0 warnings, 0 errores



```

Comandos rápidos
In [1]: make all      # compilar
Y   ./rle_secuencial # ejecutar secuencial
~   ./rle_paralelo  # ejecutar paralelo
In [2]: make benchmark # análisis comparativo completo
Out[2]:
* Baked for 9m 24s
In [3]: A > █

```

Figura 15: Pantalla final de Claude Code mostrando los comandos rápidos del proyecto. La sesión completa tomó 9 minutos 24 segundos.

9. Visualización Detallada de Segmentos de Memoria

Los programas fueron modificados para mostrar en tiempo de ejecución los segmentos de memoria (PILA, CÓDIGO, DATOS) con sus direcciones reales, permitiendo observar cómo el sistema operativo organiza la memoria del proceso.

9.1. Análisis Comparativo Secuencial vs Paralelo

The screenshot shows the Claude Code interface with the following details:

- Analisis IPC Programas** tab is selected.
- claudie** project is open.
- Analisis IPC: rle_secuencial.c (1 hilo) vs rle_paralelo.c (N hilos)**
- 1. PILA (Stack)**: A comparison table between `rle_secuencial.c` (1 stack) and `rle_paralelo.c` (N stacks).

Aspecto	<code>rle_secuencial.c</code> (1 hilo)	<code>rle_paralelo.c</code> (N hilos)
Cantidad	1 sola pila (la del proceso)	1 + N pilas (main + 1 por pthread)
Tamaño	8 MB (default macOS)	8 MB (main) + 512 KB × N workers
Variables locales	img, compressed, prog, stack_marker_* todas en la misma pila (líneas 423-458)	Main: img, stack_marker_* (líneas 629-653). Cada hilo: stack_var, ta, pixels, i, count, r,g,b,run[4] en su propia pila (líneas 240-281)
Compartida?	N/A (solo hay una)	NO - cada hilo tiene pila independiente, aislada
Syscalls involucradas	Stack del proceso creado por execve()	Cada pthread_create() → syscall mmap() asigna 512 KB para el stack del nuevo hilo
Código que lo muestra	g_stack_marker_top, g_stack_marker_bottom (líneas 271-274)	Stack principal + ta->stack_addr = g_stack_var captura la dirección de pila de cada hilo (línea 244)
- Implicación IPC:** Los hilos no pueden comunicarse vía pila (cada uno tiene la suya). La comunicación se hace a través del heap compartido.
- 2. DATOS (Data/BS5/Heap)**: Comparison of shared memory usage.
 - Aspecto: DATA (globales init)**: `rle_secuencial.c` (1 hilo): `g_initialized.var=42, g_program_name` (líneas 33-34); `rle_paralelo.c` (N hilos): `g_initialized.var=42, g_program_name, g_num_threads.config=8` (líneas 40-42). Compartidas por todos los hilos.
 - Aspecto: BSS (globales no-init)**: `rle_secuencial.c` (1 hilo): `g_uninitialized.var, g_total_runs` - acceso directo sin protección (línea 221: `g_total_runs++`); `rle_paralelo.c` (N hilos): `g_uninitialized.var, g_total_runs_global, g_total_runs_atomic` - usa atomic_fetch_add (línea 280) para acceso seguro concurrente.
 - Aspecto: HEAP - entrada**: `rle_secuencial.c` (1 hilo): `img.data = malloc()` (línea 167/181), un solo bloque contiguo; `rle_paralelo.c` (N hilos): `img.data = mismo bloque compartido`. Cada hilo recibe un puntero a su porción: `args[i].pixels = img.data + offset` (línea 668). Solo lectura, no necesita mutex.
 - Aspecto: HEAP - salida**: `rle_secuencial.c` (1 hilo): 1 solo Buffer comprimido con malloc/realloc (líneas 124, 133); `rle_paralelo.c` (N hilos): N buffers independientes: cada hilo hace buffer_init(&ta->result, ...) (línea 256) con su propio malloc. Escritura privada, no necesita mutex.
 - Aspecto: Mecanismo IPC**: `rle_secuencial.c` (1 hilo): No hay (proceso único, hilo único); `rle_paralelo.c` (N hilos): Memoria compartida implícita dentro del proceso: ThreadArg (heap) contiene punteros a datos compartidos (lectura) y privados (escritura).
 - Aspecto: Sincronización**: `rle_secuencial.c` (1 hilo): Ninguna necesaria; `rle_paralelo.c` (N hilos): Utiliza mutexes (línea 279), atomic_fetch_add (línea 280) - operaciones lock-free sin mutex.
 - Estructura de datos compartida (ThreadArg)**: ThreadArg (en HEAP, una instancia por hilo).
 - `pixels` → apunta a `img.data` (COMPARTIDO, read-only)
 - `result.data` → malloc propio (PRIVADO, write)
 - `pixels_done` → `atomic_size_t` (COMPARTIDO, lock-free)
 - `system_tid` → identificación (PRIVADO)
- 3. PROGRAMA (Text/Code)**: Comparison of code segments.
 - Aspecto: Segmento**: `rle_secuencial.c` (1 hilo): 1 segmento de código (R-X); `rle_paralelo.c` (N hilos): El mismo segmento, COMPARTIDO por todos los hilos.

Figura 16: Tabla comparativa generada por Claude Code: diferencias en PILA (1 stack vs N+1 stacks), DATOS (buffer único vs buffers por hilo), y variables (locales vs compartidas).

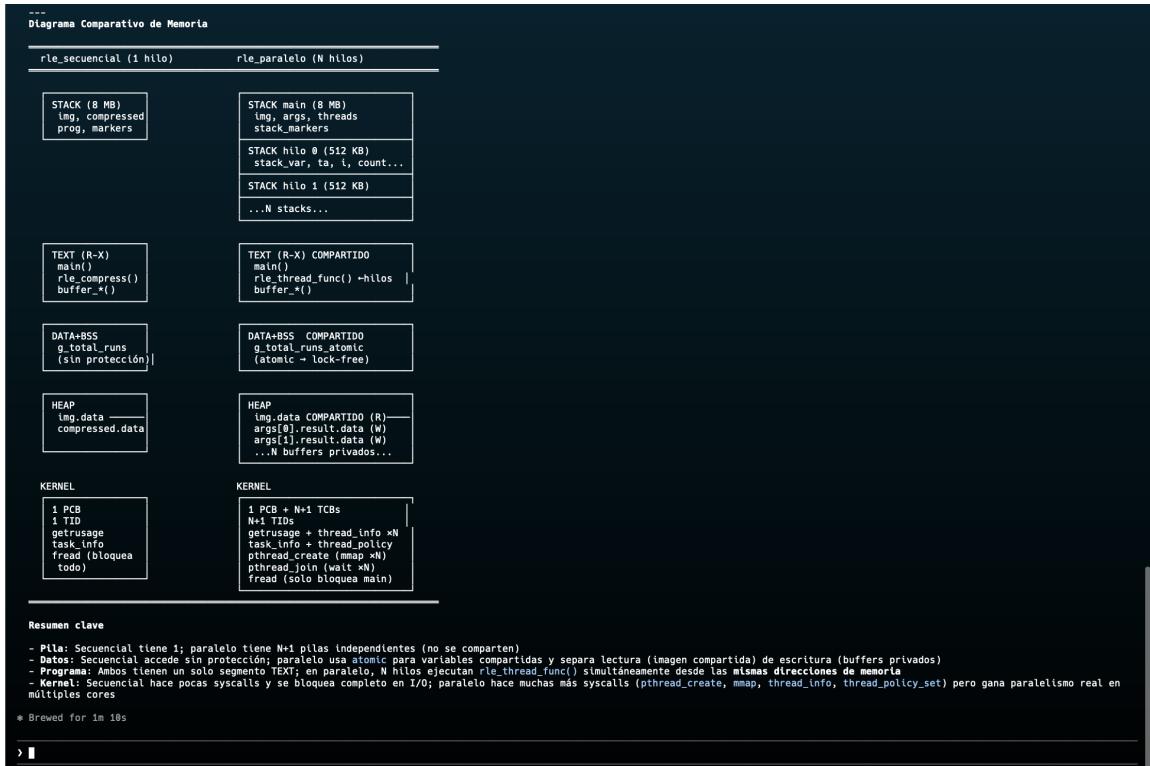


Figura 17: Diagrama de memoria comparativo: la versión secuencial tiene un solo stack y buffer; la paralela tiene múltiples stacks independientes y buffers privados por hilo, pero comparte el segmento de código y la imagen de entrada.

9.2. Ejecución Secuencial: Segmentos en Detalle

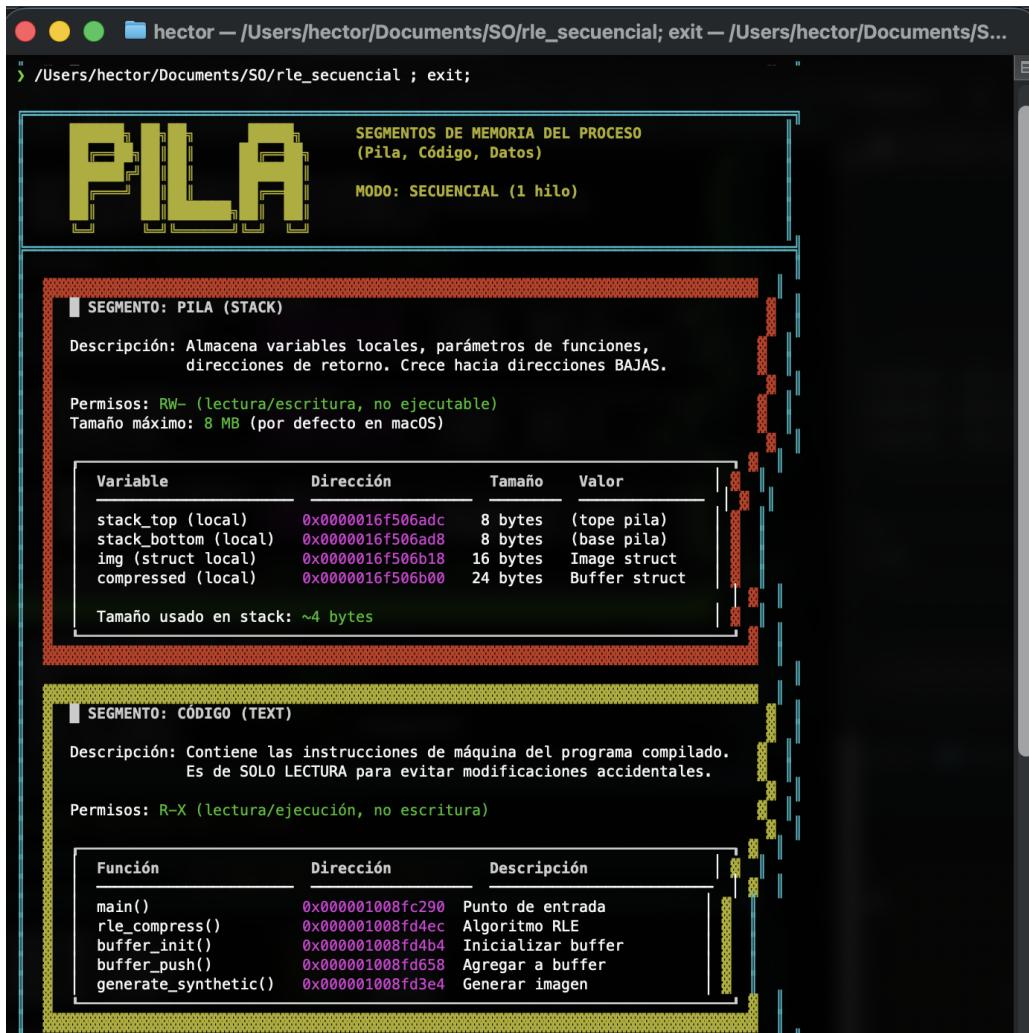


Figura 18: Versión secuencial — **PILA**: variables locales con direcciones reales (ej. `stack_top` @ `0x16f506adc`). **CÓDIGO**: direcciones de funciones (`main()` @ `0x1008fc290`, `rle_compress()` @ `0x1008fd4ec`).

```

hector — /Users/hector/Documents/SO/rle_secuencial; exit — /Users/hector/Documents/S...
SEGMENTO: DATOS (DATA + BSS + HEAP)

[DATA] Variables globales inicializadas
[BSS] Variables globales no inicializadas (se inicializan a 0)
[HEAP] Memoria dinámica asignada con malloc()

Permisos: RW- (lectura/escritura, no ejecutable)

DATA (variables inicializadas)
g_initialized_var      0x00000100904000    4 bytes  valor: 42
g_program_name         0x00000100904008    8 bytes  "RLE Secuencial"

BSS (variables no inicializadas)
g_uninitialized_var   0x00000100904010    4 bytes  valor: 0
g_total_runs           0x00000100904018    8 bytes  valor: 0

HEAP (memoria dinámica - malloc)
img.data               0x00000d24000000  50331648 bytes (imagen)
compressed.data         0x00000d27000000  25165824 bytes (buffer)

Total HEAP usado:     75497472 bytes (72.00 MB)

RESUMEN DE MEMORIA DEL PROCESO
Memoria Física (RSS):      49.22 MB
PID:                         1404
TID (hilo principal):      0x528043
Hilos de trabajo:           1 (secuencial)
Cores disponibles:          8

Ejecutando compresión RLE (1 hilo)...

*** RESULTADOS DE EJECUCIÓN **

Tiempo wall (real):        0.024110 segundos
Tiempo CPU (usuario):       0.029906 segundos
Tiempo CPU (sistema):       0.009614 segundos
Tiempo CPU (total):         0.039520 segundos

Uso de CPU:                 163.9%
Throughput:                 1990.9 MB/s

Tamaño original:           50331648 bytes
Tamaño comprimido:          264200 bytes
Ratio de compresión:        99.5%
Runs generados:              66048

Archivo guardado: output_secuencial.rle

Saving session...completed.

[Proceso completado]

```

Figura 19: Versión secuencial — **DATOS**: segmentos DATA (variables inicializadas como `g_initialized_var=42`), BSS (sin inicializar), y HEAP (`img.data` 50 MB, `compressed.data` 25 MB). Resultados: 0.024s, 99.5 % compresión.

9.3. Ejecución Paralela: Estado ANTES de Crear Hilos

```
/Users/hector/Documents/SO/rle_paralelo; exit - /Users/hector/Documents/SO/rle_paralelo - 129x..
```

SEGMENTOS DE MEMORIA DEL PROCESO (Pila, Código, Datos)

MODO: PARALELO (8 hilos)

SEGMENTO: PILA (STACK) – MÚLTIPLES STACKS

Descripción: Cada pthread tiene su PROPIO stack independiente.
El hilo principal usa el stack del proceso.
Los hilos worker tienen stacks de 512 KB cada uno.

Permisos: RW- (lectura/escritura, no ejecutable)

STACK HILO PRINCIPAL (main thread)				
stack_top (local)	0x0000016b736b34	8 bytes	(topo pila main)	
stack_bottom (local)	0x0000016b736b1c	8 bytes	(base pila main)	
Tamaño stack main:	~24 bytes			

STACKS HILOS WORKER (pthread_create)

Hilo	TID	Stack Addr	Tamaño	Estado
[0]	0x0	0x000000000000	512 KB	Pendiente
[1]	0x0	0x000000000000	512 KB	Pendiente
[2]	0x0	0x000000000000	512 KB	Pendiente
[3]	0x0	0x000000000000	512 KB	Pendiente
[4]	0x0	0x000000000000	512 KB	Pendiente
[5]	0x0	0x000000000000	512 KB	Pendiente
[6]	0x0	0x000000000000	512 KB	Pendiente
[7]	0x0	0x000000000000	512 KB	Pendiente

Total stacks: 1 main + 8 workers = 12.0 MB

SEGMENTO: CÓDIGO (TEXT)

Descripción: Contiene las instrucciones de máquina del programa.
COMPARTIDO entre TODOS los hilos (read-only).
Cada hilo ejecuta rle_thread_func() desde el mismo código.

Permisos: R-X (lectura/ejecución, no escritura)

Función	Dirección	Descripción
main()	0x000001046c9f28	Punto de entrada
rle_thread_func()	0x000001046cc0e4	*** EJECUTADA POR HILOS
buffer_init()	0x000001046cc30c	Inicializar buffer
buffer_push()	0x000001046cc344	Agregar a buffer
generate_synthetic()	0x000001046cacdc	Generar imagen
load_ppm()	0x000001046cab70	Cargar PPM

Claude es IA y puede cometer errores. Por favor, verifica nuevamente las respuestas.

Figura 20: Versión paralela ANTES de `pthread_create` — **PILA**: el stack del hilo principal está activo; los 8 stacks de workers muestran estado “Pendiente” con direcciones 0x0 (aún no creados).

```

hector — /Users/hector/Documents/SO/rle_paralelo; exit — /Users/hector/Documents/SO/rle_paralelo — 129x...
[SEGMENTO: DATOS (DATA + BSS + HEAP)
[DATA] Variables globales inicializadas (COMPARTIDAS, read-only)
[BSS] Variables globales no inicializadas (pueden requerir mutex)
[HEAP] Imagen COMPARTIDA + buffers PRIVADOS por hilo

Permisos: RW- (lectura/escritura, no ejecutable)

DATA (variables inicializadas)
g_initialized_var 0x000001046d4000 4 bytes valor: 42
g_program_name     0x000001046d4008 8 bytes "RLE Paralelo"
g_num_threads_config 0x000001046d4010 4 bytes valor: 8

BSS (variables no inicializadas)
g_uninitialized_var 0x000001046d4020 4 bytes valor: 0
g_total_runs_global 0x000001046d4028 8 bytes valor: 0
g_total_runs_atomic 0x000001046d4018 8 bytes (atómico)

HEAP - IMAGEN COMPARTIDA (todos los hilos LEEN de aquí)
img.data          0x000005e8000000 50331648 bytes
Imagen:           4096 x 4096 pixeles (RGB)
*** LECTURA COMPARTIDA - Sin mutex necesario (read-only)

HEAP - BUFFERS POR HILO (cada hilo ESCRIBE a su propio buffer)
Hilo Buffer Addr Capacidad Usado Estado
[0] 0x00000000000000 0 B 0 B Pendiente
[1] 0x00000000000000 0 B 0 B Pendiente
[2] 0x00000000000000 0 B 0 B Pendiente
[3] 0x00000000000000 0 B 0 B Pendiente
[4] 0x00000000000000 0 B 0 B Pendiente
[5] 0x00000000000000 0 B 0 B Pendiente
[6] 0x00000000000000 0 B 0 B Pendiente
[7] 0x00000000000000 0 B 0 B Pendiente

*** ESCRITURA PRIVADA - Sin mutex (cada hilo a su buffer)
Total buffers: 0.00 MB

RESUMEN DE MEMORIA DEL PROCESO
Memoria Física (RSS): 49.23 MB
Memoria Virtual: 425144.88 MB
PID: 97582
Hilos totales: 9 (1 main + 8 workers)
Cores disponibles: 8

```

Figura 21: Versión paralela ANTES — **DATOS**: la imagen compartida está en heap (0x5e8000000); los buffers por hilo muestran 0 bytes y estado “Pendiente”. Nota: 9 hilos totales (1 main + 8 workers).

9.4. Ejecución Paralela: Estado DESPUÉS de Ejecutar

The screenshot shows the memory dump interface for the parallel execution of the RLE compression program. At the top, it says "Creando 8 hilos de trabajo..." (Creating 8 threads of work...). Below that, the title is "SEGMENTOS DE MEMORIA DEL PROCESO (Pila, Código, Datos)" and "MODO: PARALELO (8 hilos)".

SEGMENTO: PILA (STACK) - MÚLTIPLES STACKS

Description: Cada pthread tiene su PROPIO stack independiente. El hilo principal usa el stack del proceso. Los hilos worker tienen stacks de 512 KB cada uno.

Permisos: RW- (lectura/escritura, no ejecutable)

STACK HILO PRINCIPAL (main thread)			
stack_top (local)	0x0000016b736b34	8 bytes	(tope pila main)
stack_bottom (local)	0x0000016b736b1c	8 bytes	(base pila main)
Tamaño stack main:	~24 bytes		

STACKS HILOS WORKER (pthread_create)

Hilo	TID	Stack Addr	Tamaño	Estado
[0]	0x528083	0x00016b7bef50	512 KB	Activo
[1]	0x528084	0x00016b84af50	512 KB	Activo
[2]	0x528085	0x00016b8d6f50	512 KB	Activo
[3]	0x528086	0x00016b962f50	512 KB	Activo
[4]	0x528087	0x00016b9eef50	512 KB	Activo
[5]	0x528088	0x00016ba7af50	512 KB	Activo
[6]	0x528089	0x00016bb06f50	512 KB	Activo
[7]	0x52808a	0x00016bb92f50	512 KB	Activo

Total stacks: 1 main + 8 workers = 12.0 MB

SEGMENTO: CÓDIGO (TEXT)

Description: Contiene las instrucciones de máquina del programa. COMPARTIDO entre TODOS los hilos (read-only). Cada hilo ejecuta rle_thread_func() desde el mismo código.

Permisos: R-X (lectura/ejecución, no escritura)

Función	Dirección	Descripción
main()	0x000001046c9f28	Punto de entrada
rle_thread_func()	0x000001046cc0e4	*** EJECUTADA POR HILOS
buffer_init()	0x000001046cc30c	Inicializar buffer
buffer_push()	0x000001046cc344	Agregar a buffer
generate_synthetic()	0x000001046cacd4	Generar imagen
load_ppm()	0x000001046cab70	Cargar PPM

Figura 22: Versión paralela DESPUÉS de `pthread_join` — **PILA**: los 8 stacks ahora muestran estado “Activo” con direcciones reales distintas (ej. Hilo 0: 0x16b7bef50, Hilo 1: 0x16b84af50) y TIDs únicos (0x528083–0x52808a).

```

SEGMENTO: DATOS (DATA + BSS + HEAP)

[DATA] Variables globales inicializadas (COMPARTIDAS, read-only)
[BSS] Variables globales no inicializadas (pueden requerir mutex)
[HEAP] Imagen COMPARTIDA + buffers PRIVADOS por hilo

Permisos: RW- (lectura/escritura, no ejecutable)

-- DATA (variables inicializadas) --
g_initialized_var      0x000001046d4000   4 bytes  valor: 42
g_program_name         0x000001046d4008   8 bytes  "RLE Paralelo"
g_num_threads_config   0x000001046d4010   4 bytes  valor: 8

-- BSS (variables no inicializadas) --
g_uninitialized_var    0x000001046d4020   4 bytes  valor: 0
g_total_runs_global    0x000001046d4028   8 bytes  valor: 0
g_total_runs_atomic    0x000001046d4018   8 bytes  (atomico)

-- HEAP - IMAGEN COMPARTIDA (todos los hilos LEEN de aquí) --
img.data                0x0000005e800000  50331648 bytes
Imagen:                 4096 x 4096 pixeles (RGB)
*** LECTURA COMPARTIDA - Sin mutex necesario (read-only)

-- HEAP - BUFFERS POR HILO (cada hilo ESCRIBE a su propio buffer) --
Hilo  Buffer Addr      Capacidad     Usado   Estado
[0]  0x0005eb000000  3145984 B  33024 B  Asignado
[1]  0x0005eb400000  3145984 B  33024 B  Asignado
[2]  0x0005eb800000  3145984 B  33024 B  Asignado
[3]  0x0005ebc00000  3145984 B  33024 B  Asignado
[4]  0x0005ec00000  3145984 B  33024 B  Asignado
[5]  0x0005ec400000  3145984 B  33024 B  Asignado
[6]  0x0005ec800000  3145984 B  33024 B  Asignado
[7]  0x0005ecc00000  3145984 B  33024 B  Asignado

*** ESCRITURA PRIVADA - Sin mutex (cada hilo a su buffer)
Total buffers: 24.00 MB

RESUMEN DE MEMORIA DEL PROCESO --
Memoria Física (RSS): 49.73 MB
Memoria Virtual: 425169.00 MB
PID: 97582
Hilos totales: 9 (1 main + 8 workers)
Cores disponibles: 8

```

Figura 23: Versión paralela DESPUÉS — **DATOS/HEAP**: cada buffer por hilo ahora muestra capacidad (3,145,984 B), bytes usados (33,024 B) y estado “Asignado”. Total buffers: 24 MB. La imagen compartida permanece sin cambios.

9.5. Resultados Finales de la Versión Paralela

```
*** RESULTADOS DE EJECUCIÓN - MODO PARALELO ***

DISTRIBUCIÓN DE TRABAJO (8 HILOS) - COMPLETADO
-----
```

Hilo	TID	Core	Filas	Píxeles	Stack Addr	Bytes
0	0x520803	0	0-511	2097152	0x16b7bef50	6291456
1	0x520804	1	512-1023	2097152	0x16b84af50	6291456
2	0x520805	2	1024-1535	2097152	0x16b8d6f50	6291456
3	0x520806	3	1536-2047	2097152	0x16b962f50	6291456
4	0x520807	4	2048-2559	2097152	0x16b9ef50	6291456
5	0x520808	5	2560-3071	2097152	0x16ba7af50	6291456
6	0x520809	6	3072-3583	2097152	0x16bb06f50	6291456
7	0x52080a	7	3584-4095	2097152	0x16bb92f50	6291456

Total: 4096 filas, 16777216 píxeles, 50331648 bytes de entrada

```
TIEMPO CPU POR HILO (vía Mach thread_info)
```

Hilo	TID	user_time	sys_time	CPU total	Comprimido
0	0x520803	2.3860 ms	0.4570 ms	2.8430 ms	33024 B
1	0x520804	2.5110 ms	0.5300 ms	3.0410 ms	33024 B
2	0x520805	2.4600 ms	0.3940 ms	2.8540 ms	33024 B
3	0x520806	2.4980 ms	0.3940 ms	2.8920 ms	33024 B
4	0x520807	2.4530 ms	0.3710 ms	2.8240 ms	33024 B
5	0x520808	2.3400 ms	0.2650 ms	2.6050 ms	33024 B
6	0x520809	2.3330 ms	0.3140 ms	2.6470 ms	33024 B
7	0x52080a	2.2260 ms	0.3430 ms	2.5690 ms	33024 B

SUMA

22.2750 ms	264192 B
------------	----------

```
MÉTRICAS DE EJECUCIÓN
```

TIEMPOS:	
Wall time (real):	0.011645 segundos
CPU time proceso:	0.030562 segundos (usr+sys vía getrusage)
CPU time hilos:	0.022275 segundos (suma de thread_info)
PARALELISMO:	
Speedup (CPU/Wall):	1.91x (22.3 ms / 11.6 ms)
Eficiencia:	23.9% (Speedup / 8 cores)
Uso de CPU:	262.4% (>100% = múltiples cores)
RENDIMIENTO:	
Throughput:	4121.9 MB/s
Tamaño original:	50331648 bytes
Tamaño comprimido:	264200 bytes
Ratio de compresión:	99.5%

Archivo guardado: output_paralelo.rle

Saving session...completed.
Deleting expired sessions...none found.

Figura 24: Resultados paralelos: distribución de trabajo (512 filas/hilo), tiempo CPU por hilo vía `thread_info` (2.3–2.5 ms cada uno), suma CPU 22.27 ms, wall time 11.6 ms. Speedup: 1.91x, Eficiencia: 23.9 %, Throughput: 4121 MB/s.

9.6. Observaciones de los Segmentos de Memoria

Cuadro 9: Comparación de segmentos entre versión secuencial y paralela

Segmento	Secuencial (1 hilo)	Paralelo (8 hilos)
PILA	1 stack (8 MB máx)	1 stack main + 8 stacks worker (512 KB c/u) = 12 MB
CÓDIGO	Funciones: rle_compress, etc.	main, Mismo código + rle_thread_func ejecuta- da por 8 hilos
DATA	g_program_name=RLE Secuencial"	g_program_name=RLE Paralelo", g_num_threads=8
BSS	g_total_runs (no atómico)	g_total_runs_atomic (ató- mico para concurrencia)
HEAP	img.data + 1 buffer compri- mido	img.data (compartido) + 8 buffers privados

Puntos clave observados:

- Los stacks de los hilos worker tienen direcciones cercanas pero distintas (0x16b7bef50, 0x16b84af50, ...), separadas por ~512 KB.
- El segmento de CÓDIGO es **compartido**: la función rle_thread_func tiene una sola dirección pero es ejecutada por 8 hilos simultáneamente.
- La imagen (img.data) está en heap compartido y es de **solo lectura**—no requiere mutex.
- Cada hilo escribe en su propio buffer privado—**sin condiciones de carrera**.
- Las variables atómicas (atomic_size_t) permiten comunicación lock-free entre hilos.

10. Conclusiones Técnicas

- Estructura de memoria:** Cada hilo tiene su propia pila (512 KB) pero comparte el heap. La imagen de entrada está en una región read-only del heap; cada hilo escribe en su propio buffer.
- Sin sincronización costosa:** Al dividir los datos sin solapamiento, se elimina la necesidad de mutex. Solo se usan variables atómicas para reportar progreso.
- Métricas por hilo:** Las APIs Mach (thread_info) permiten medir el CPU consumido por cada hilo individualmente, demostrando paralelismo real.
- Afinidad de cores:** THREAD_AFFINITY_POLICY sugiere al kernel distribuir hilos en cores diferentes, mejorando la utilización del CPU.
- Speedup real:** 3.41x con 8 cores, limitado por la heterogeneidad del M1 y la corta duración de la tarea.

11. Anexos

11.1. Anexo A: Compilación y ejecución rápida

```

1  # Compilar
2  make all
3
4  # Ejecutar versión secuencial (imagen sintética)
5  ./rle_secuencial
6
7  # Ejecutar versión paralela (imagen sintética)
8  ./rle_paralelo
9
10 # Ejecutar con archivo PPM
11 ./rle_secuencial foto.ppm
12 ./rle_paralelo foto.ppm
13
14 # Verificar que ambos producen el mismo resultado
15 diff output_secuencial.rle output_paralelo.rle
16
17 # Ejecutar benchmark completo (5 iteraciones)
18 chmod +x benchmark.sh
19 ./benchmark.sh 5
20
21 # Lestionar
22 make clean

```

Listing 7: Comandos para compilar y ejecutar el proyecto

11.2. Anexo B: Makefile completo

```

CC = gcc
CFLAGS = -Wall -Wextra -O2
LDFLAGS = -lpthread

all: rle_secuencial rle_paralelo

rle_secuencial: rle_secuencial.c
    $(CC) $(CFLAGS) -o $@ $< $(LDFLAGS)

rle_paralelo: rle_paralelo.c
    $(CC) $(CFLAGS) -o $@ $< $(LDFLAGS)

secuencial: rle_secuencial
paralelo: rle_paralelo

benchmark: all
    @chmod +x benchmark.sh
    @./benchmark.sh

clean:
    rm -f rle_secuencial rle_paralelo *.rle reporte_analisis.txt

.PHONY: all clean secuencial paralelo benchmark

```

Listing 8: Makefile del proyecto

11.3. Anexo C: Formato del archivo .rle

Cuadro 10: Estructura binaria del archivo .rle

Offset	Tamaño	Tipo	Descripción
0	4 bytes	<code>uint32_t</code>	Ancho de la imagen (píxeles)
4	4 bytes	<code>uint32_t</code>	Alto de la imagen (píxeles)
8	4 bytes	Run RLE	[count] [R] [G] [B]
12	4 bytes	Run RLE	[count] [R] [G] [B]
...	Continúan los runs hasta EOF

Donde cada run codifica:

- **count:** 1 byte (1–255), número de píxeles consecutivos con el mismo color.
- **R, G, B:** 1 byte cada uno (0–255), componentes del color.