SUBMIT VII:

Ana Gabriela Argüello, Laura Isabel Olivero, Kevin Pérez y Juan Diego Rojas

1. **ANALISIS ASINTOTICO:**

Punto 1 secuencial:

- La función sortRow tiene una complejidad $O(n^2 \log n)$ porque el sort tiene una complejidad $O(n \log n)$ y esto se multiplica por todas las filas
- La función sortColumn tiene una complejidad $O(n^2 \log n)$ porque el sort tiene una complejidad $O(n \log n)$ y esto se multiplica por todas las filas
- La función shearSortParallel tiene una complejidad $O(n^2 \log^2 n)$ porque en cada llamamiento ordena todas las filas y todas las columnas.

La complejidad temporal es $O(n^2 \log^2 n)$.

- La matriz ocupa $O(n^2)$.
- El vector temporal usa $O(n)$

La complejidad espacial total es $S(n) = O(n^2)$.

Punto 1 paralelo:

- La función sortRow tiene una complejidad $O(n^2 \log \frac{n}{p})$ porque cada fila se puede ordenar en paralelo. El ordenamiento de una fila tiene costo $O(n \log n)$ y, al distribuir las filas entre p hilos, el costo total se reduce proporcionalmente al número de procesadores.
- La función sortColumn tiene una complejidad $O(n^2 \log \frac{n}{p})$, ya que cada columna también puede ordenarse en paralelo con el mismo principio anterior.
- La función shearSortParallel tiene una complejidad $O((n^2 \log n)/p)$, porque en cada iteración ordena todas las filas y columnas, y este proceso se repite $O(\log n)$ veces, aprovechando la ejecución concurrente en los hilos.

La complejidad temporal es $O((n^2 \log n)/p)$, donde p representa el número de procesadores o hilos disponibles.

La matriz ocupa $O(n^2)$.

El vector temporal usa O(n).

La complejidad espacial total es $S(n) = O(n^2)$.

Punto 2 secuencial:

- La función binarySearch tiene una complejidad $O(n \log n)$ porque en cada iteración del ciclo while reduce el espacio de búsqueda a la mitad, dividiendo el rango entre dos hasta encontrar el elemento o agotar el arreglo
- En el main, la creación del vector y su inicialización tienen una complejidad $O(n)$, ya que se recorren todos los elementos para asignarles un valor.

La complejidad total del programa está dominada por la inicialización del arreglo, por lo que la complejidad temporal es O(n). Si se considera únicamente la búsqueda, la complejidad es O(log n).

- La estructura de datos principal es el vector, que ocupa O(n) posiciones en memoria.
- No se crean estructuras adicionales significativas.

La complejidad espacial total es S(n) = O(n).

Punto 2 paralelo:

- La función parallelBinarySearch tiene una complejidad $O((n \log n)/p)$ en el mejor de los casos, ya que el arreglo se divide entre p hilos, y cada hilo realiza una búsqueda binaria independiente sobre su segmento. La búsqueda binaria mantiene su comportamiento logarítmico dentro de cada porción.
- En el peor de los casos, todos los hilos deben ejecutar su búsqueda antes de que uno encuentre el elemento, lo que mantiene una complejidad total aproximada de O(log n), pero con una reducción práctica del tiempo debido a la ejecución paralela.
- La inicialización del vector en el main tiene una complejidad O(n), ya que se recorre todo el arreglo para asignar valores secuenciales.

La complejidad temporal total es O(n / p + log n), dominada por la carga de datos y el beneficio parcial del paralelismo en la búsqueda.

- La estructura de datos principal es el vector, que ocupa O(n) en memoria.
- No se crean estructuras adicionales significativas.

La complejidad espacial total es S(n) = O(n).

2. Speedup y Métricas de Walltime

Para realizar el cálculo del speedup, se añadieron a los códigos instrucciones que medían el tiempo. Primero se procederá a mostrar los resultados encontrados de los tiempos de ejecución (wall time).

a. Shearsort Secuencial:

Se realizaron tres ejecuciones del código, y los walltimes arrojados fueron los siguientes:

- Ejecución 1: 7.0606e-05 segundos
- Ejecución 2: 1.8802e-05 segundos
- Ejecución 3: 3.1103e-05 segundos

Para un tiempo de ejecución promedio de 0.0000401703 segundos.

b. Shearsort Paralelo:

Se realizaron tres ejecuciones del código, y los walltimes arrojados fueron los siguientes:

- Ejecución 1: 0.00467596 segundos
- Ejecución 2: 0.00180496 segundos
- Ejecución 3: 0.00383026 segundos

Para un tiempo de ejecución promedio de 0.00343706 segundos.

c. Búsqueda Binaria Secuencial:

Se realizaron tres ejecuciones del código, y los walltimes arrojados fueron los siguientes:

- Ejecución 1: 8.501e-06 segundos
- Ejecución 2: 2.201e-06 segundos
- Ejecución 3: 8e-07 segundos

Para un tiempo de ejecución promedio de 0.000003834 segundos.

d. Búsqueda Binaria Paralela:

Se realizaron tres ejecuciones del código, y los walltimes arrojados fueron los siguientes:

- Ejecución 1: 0.00041966 segundos
- Ejecución 2: 0.000511474 segundos
- Ejecución 3: 0.000577384 segundos

Para un tiempo de ejecución promedio de 0.0005028393 segundos.

Con estos resultados, se procede a realizarse el cálculo del speedup para cada caso:

a. Speedup Shearsort:

- Tiempo de la ejecución secuencial: 0.0000401703
- Tiempo de la ejecución paralela: 0.00343706

$$s = \frac{0.0000401703}{0.00343706} = 0.0116874$$

b. Speedup Búsqueda Binaria:

- Tiempo de la ejecución secuencial: 0.0000401703
- Tiempo de la ejecución paralela: 0.00343706

$$s = \frac{0.000003834}{0.0005028393} = 0.0076247$$

En ambos casos, el speedup fue menor que 1 (es decir, la versión secuencial resultó más eficiente que la paralela), seguramente debido al tamaño de los arreglos evaluados, dado que al ser pequeños, el algoritmo tarda más tiempo organizando los hilos que lo que realmente se ahorra haciendo los cálculos. Si se implementara el ejercicio sobre un arreglo de tamaño más elevado, seguramente el paralelismo resultaría más efectivo.