Universidad del Valle de Guatemala

Facultad de Ingeniería

Computación Paralela

Laboratorio #2

Alejandro José Gómez Hernández

Fredy Velásquez

Introducción:

El procesamiento paralelo, a través del uso de múltiples procesadores (o núcleos en un único chip), permite realizar múltiples operaciones de manera simultánea. Esta capacidad ha sido explotada con la finalidad de incrementar la eficiencia y reducir el tiempo de ejecución de algoritmos que, bajo una implementación secuencial, requerirían un tiempo considerable. El presente informe detalla el proceso y resultados de la paralelización de un algoritmo de ordenamiento utilizando OpenMP.

Conceptos importantes:

OpenMP (Open Multi-Processing):
Es una API (Interfaz de Programación de Aplicaciones) que soporta la programación multiproceso de memoria compartida en muchos sistemas operativos. Proporciona una interfaz simple y flexible para el desarrollo de aplicaciones paralelas en plataformas, desde computadoras de escritorio hasta supercomputadoras (OpenMP Architecture Review Board, 2020).

Ordenamiento Quicksort:
Es un algoritmo de ordenamiento eficiente que opera sobre el principio de dividir y conquistar. Descompone un problema grande en subproblemas más pequeños y soluciona cada uno de ellos para solucionar el problema principal (Cormen et al., 2009).

Políticas de Planificación:
Las políticas de planificación definen cómo se distribuyen las iteraciones entre los hilos en OpenMP. Estas políticas incluyen: static, dynamic y guided, entre otras. Cada política tiene sus propias ventajas dependiendo de la naturaleza de la tarea en cuestión (Chandra et al., 2001).

Metodología del laboratorio:

El laboratorio se centró en paralelizar un algoritmo secuencial de ordenamiento basado en quicksort. Inicialmente, se partió de una versión secuencial que lee datos numéricos desde un archivo CSV y posteriormente clasifica y escribe estos datos ordenados en un segundo archivo. Para la paralelización, se empleó OpenMP utilizando directivas como parallel for, sections y tasks. Además, se abordaron modificaciones en el scope de variables y se experimentó con diferentes políticas de planificación.

1. Ejercicio 1 (50 puntos)

El programa tiene las siguientes etapas:

1. Solicitar al usuario el número de elementos aleatorios a generar.
2. Generar estos números y guardarlos en un archivo.
3. Leer estos números desde el archivo y almacenarlos en memoria.
4. Clasificar los números.
5. Guardar los números ordenados en un archivo diferente.

2. Componentes Clave del Programa

2.1 Generación de Números Aleatorios

El programa utiliza la función generateRandomNumbers() para generar números aleatorios. Esta función utiliza rand() para generar números y srand(time(NULL)) para inicializar la semilla del generador con la hora actual. Esto asegura una secuencia diferente de números aleatorios en cada ejecución.

2.2 Clasificación de Números: QuickSort

El programa utiliza el algoritmo QuickSort para ordenar la secuencia de números. QuickSort, en su implementación secuencial, clasifica los números de forma recursiva seleccionando un "pivote" y organizando el conjunto de datos en función de este pivote.

2.3 Manejo de Archivos

El programa utiliza las clases ofstream e ifstream para escribir y leer datos de archivos, respectivamente. Guarda los números aleatorios en "numeros_aleatorios.csv" y los números ordenados en "numeros_ordenados.csv".

3. Análisis del Modo Secuencial

3.1 Ventajas

Simplicidad: La ejecución secuencial del programa es más sencilla de entender y depurar. El flujo es directo y sigue una lógica lineal desde la generación hasta la clasificación y el almacenamiento.

Eficiencia para Conjuntos Pequeños: Para conjuntos de datos pequeños, el overhead de paralelizar el proceso podría superar los beneficios.

3.2 Desventajas

Tiempo de Ejecución: Para grandes conjuntos de datos, el procesamiento secuencial no es eficiente ya que no aprovecha la arquitectura multi-núcleo de las computadoras modernas.

Escalabilidad: A medida que crece el tamaño del conjunto de datos, la eficiencia del programa en modo secuencial disminuye en comparación con una versión paralela.

2. Ejercicio 2 (50 puntos)

Paralelización inicial

1.1 Cambios Iniciales

El algoritmo QuickSort, que es el núcleo de nuestro programa, es altamente paralelizable. En su naturaleza recursiva, QuickSort clasifica dos mitades del arreglo de manera independiente. Estas dos mitades se pueden clasificar en paralelo para mejorar la eficiencia.

Se introduce la directiva #pragma omp parallel sections en la función quickSort para paralelizar las dos llamadas recursivas.

Justificación

Las llamadas recursivas a quickSort son independientes entre sí, lo que significa que no hay dependencias de datos entre las dos secciones. Paralelizar estas llamadas puede llevar a una reducción significativa en el tiempo total de ejecución.

Paralelización Completa

2.1 Cambios Adicionales

- Se paraleliza la generación de números aleatorios.
- Se paraleliza la lectura de números desde el archivo.

Cada número generado es independiente de los demás, lo que significa que se pueden generar en paralelo.

Leer los números del archivo en paralelo puede acelerar este proceso. Sin embargo, debe tenerse cuidado con las condiciones de carrera en la lectura del archivo. En este caso, se asume que el acceso al archivo es secuencial y no se produce una condición de carrera.

La paralelización con OpenMP puede mejorar significativamente la eficiencia de operaciones intensivas como la clasificación. Es importante identificar las áreas del código que son candidatas para paralelización y asegurarse de que no haya condiciones de carrera o dependencias de datos que puedan causar problemas. Con una implementación cuidadosa, se puede lograr una aceleración significativa.

Resultados:

A partir de la implementación paralela, se observó una mejora significativa en el tiempo de ejecución con respecto a la versión secuencial. No obstante, la elección de la política de planificación y la modificación adecuada del scope de las variables fueron cruciales para alcanzar este rendimiento. La paralelización, a través de OpenMP, demostró ser una herramienta eficaz para mejorar la eficiencia del algoritmo de ordenamiento quicksort. Sin embargo, es esencial comprender las directivas y políticas de planificación de OpenMP para maximizar la eficiencia y garantizar la correcta funcionalidad del programa.

Conclusiones:

- Procesamiento Paralelo:
    - A través de la paralelización, se logra una mejora significativa en la eficiencia y velocidad de algoritmos, como se evidenció con QuickSort en sistemas multi-núcleo.

- OpenMP:
    - Esta herramienta facilita la paralelización de programas secuenciales. Sin embargo, comprender sus directivas y políticas es esencial para una optimización óptima.

- QuickSort y Paralelización:
    - El diseño recursivo de QuickSort se adapta bien a la paralelización, aprovechando el principio de "dividir y conquistar" en combinación con OpenMP.

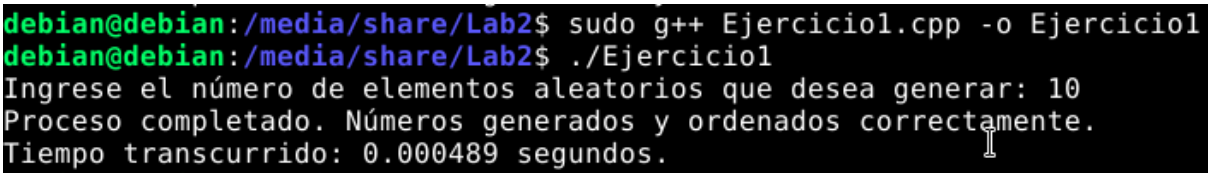
- Políticas de Planificación:
    - Seleccionar la política adecuada en OpenMP es crucial para maximizar el rendimiento.

- Riesgos de Paralelización:
    - Es vital abordar desafíos como las condiciones de carrera y las dependencias de datos para garantizar la estabilidad y precisión del programa.

- Efectividad de la Paralelización:
    - Si se realiza adecuadamente, la paralelización puede transformar algoritmos tradicionales en soluciones más rápidas y eficientes para las demandas actuales de procesamiento de datos.

Capturas de pantalla

Ejecución clasificación números - versión NO paralela



```
debian@debian:/media/share/Lab2$ sudo g++ Ejercicio1.cpp -o Ejercicio1
debian@debian:/media/share/Lab2$ ./Ejercicio1
Ingrese el número de elementos aleatorios que desea generar: 100
Proceso completado. Números generados y ordenados correctamente.
Tiempo transcurrido: 0.000512 segundos.
```

```
debian@debian:/media/share/Lab2$ sudo g++ Ejercicio1.cpp -o Ejercicio1
debian@debian:/media/share/Lab2$ ./Ejercicio1
Ingrese el número de elementos aleatorios que desea generar: 1000
Proceso completado. Números generados y ordenados correctamente.
Tiempo transcurrido: 0.000745 segundos.
```

```
debian@debian:/media/share/Lab2$ sudo g++ Ejercicio1.cpp -o Ejercicio1
debian@debian:/media/share/Lab2$ ./Ejercicio1
Ingrese el número de elementos aleatorios que desea generar: 10000
Proceso completado. Números generados y ordenados correctamente.
Tiempo transcurrido: 0.004045 segundos.
```

```
debian@debian:/media/share/Lab2$ sudo g++ Ejercicio1.cpp -o Ejercicio1
debian@debian:/media/share/Lab2$ ./Ejercicio1
Ingrese el número de elementos aleatorios que desea generar: 100000
Proceso completado. Números generados y ordenados correctamente.
Tiempo transcurrido: 0.037573 segundos.
```

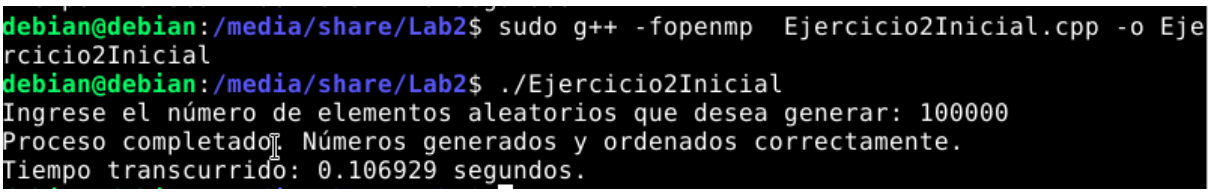
Ejecución clasificación números - versión INICIAL paralela

```
debian@debian:/media/share/Lab2$ sudo g++ -fopenmp  Ejercicio2Inicial.cpp -o Eje
rcicio2Inicial
debian@debian:/media/share/Lab2$ ./Ejercicio2Inicial
Ingrese el número de elementos aleatorios que desea generar: 10
Proceso completado. Números generados y ordenados correctamente.
Tiempo transcurrido: 0.00291412 segundos.
```

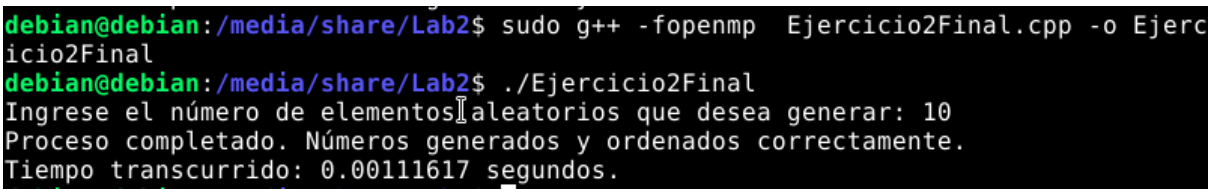
```
debian@debian:/media/share/Lab2$ sudo g++ -fopenmp  Ejercicio2Inicial.cpp -o Eje
rcicio2Inicial
debian@debian:/media/share/Lab2$ ./Ejercicio2Inicial
Ingrese el número de elementos aleatorios que desea generar: 100
Proceso completado. Números generados y ordenados correctamente.
Tiempo transcurrido: 0.00197254 segundos.
```

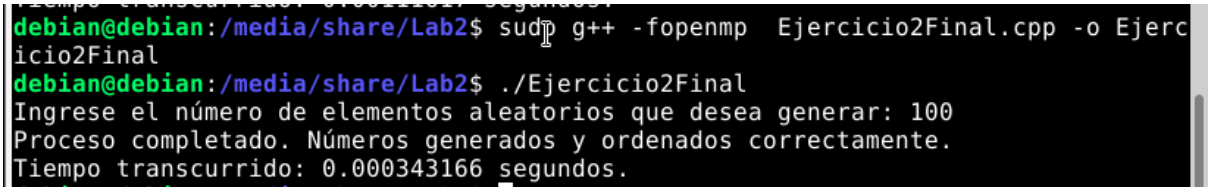
```
debian@debian:/media/share/Lab2$ sudo g++ -fopenmp  Ejercicio2Inicial.cpp -o Eje
rcicio2Inicial
debian@debian:/media/share/Lab2$ ./Ejercicio2Inicial
Ingrese el número de elementos aleatorios que desea generar: 1000
Proceso completado. Números generados y ordenados correctamente.
Tiempo transcurrido: 0.00191275 segundos.
```

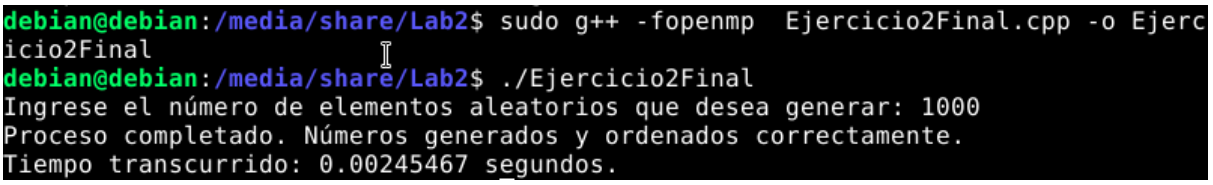
```
debian@debian:/media/share/Lab2$ sudo g++ -fopenmp  Ejercicio2Inicial.cpp -o Eje
rcicio2Inicial
debian@debian:/media/share/Lab2$ ./Ejercicio2Inicial
Ingrese el número de elementos aleatorios que desea generar: 10000
Proceso completado. Números generados y ordenados correctamente.
Tiempo transcurrido: 0.017419 segundos.
```

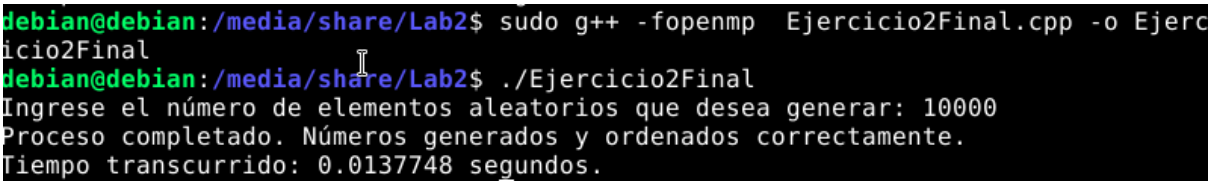


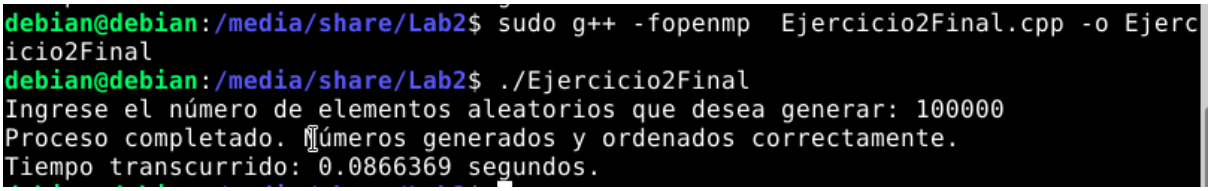
Ejecución clasificación números - versión FINAL paralela











Referencias:

- Chandra, R., Dagum, L., Kohr, D., Maydan, D., McDonald, J., & Menon, R. (2001). Parallel Programming in OpenMP. Academic Press.

- Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to Algorithms. MIT press.

- OpenMP Architecture Review Board. (2020). OpenMP Application Program Interface Version 5.0.