Taller 4 - Arquitectura SMP y SMD.

Integrantes:
Juan Manuel Perea - 1926462.
Erika Garcia - 2259395
Yenny Margot Rivas - 2182527

Presentado a:
PhD. Manuel Alejandro Pastrana.

Universidad del Valle.
Escuela de Ingeniería de Sistemas y Computación.
Infraestructuras Paralelas y Distribuidas.
Octubre de 2025.

Ejercicio 1: Arquitectura SMP (Symmetric Multiprocessing).

Para este trabajo decidimos implementar dos versiones del mismo algoritmo, una de forma secuencial y otra con hilos. La idea de hacerlo así fue comparar de manera clara cómo funcionan y cuánto tardan en ejecutarse. De esta manera podemos ver de forma sencilla la diferencia entre hacer todo en un solo flujo de instrucciones o repartir el trabajo en varios hilos que corren al mismo tiempo.

Código secuencial:

En la versión secuencial, primero se crea una matriz de tamaño 1000x1000 con números enteros generados de forma aleatoria usando la librería estándar de Python con *random.seed(0).* Al fijar la semilla se asegura que siempre se obtengan los mismos valores, lo que permite comparar resultados de manera justa.

Luego, se divide la matriz en bloques más pequeños de 100x100, lo que facilita trabajar con partes específicas en lugar de procesar toda la matriz de una sola vez. Cada bloque se recorre de manera secuencial en un ciclo y dentro de él se calcula la suma de todos sus elementos. Estas sumas parciales se van acumulando en una variable hasta obtener la suma total de la matriz.

Este enfoque es simple y lineal, ya que todas las operaciones se realizan de manera secuencial, un solo hilo de ejecución se encarga de ir procesando bloque por bloque hasta terminar.

Código con Hilos:

En la versión SMP con hilos, la idea general del programa es la misma, se crea una matriz de 1000x1000 con números enteros generados aleatoriamente, pero en este caso usando *np.random.seed(0)* junto con NumPy, ya que esta librería aprovecha mejor operaciones vectorizadas y procesamiento en paralelo. Esto permite que los cálculos de bloques sean más rápidos y eficientes.

Luego, se divide la matriz en bloques de 100x100, y se define una función que se encarga de calcular la suma de cada bloque. La diferencia principal es que, en lugar de recorrer los bloques uno por uno en un solo ciclo, se utiliza la librería *ThreadPoolExecutor*, que permite enviar cada bloque a un hilo diferente, tratándolo como una tarea independiente que se puede ejecutar de forma simultánea con las demás. Después, a medida de que los hilos van terminando sus cálculos, los resultados se van recuperando y acumulando en una variable para obtener nuevamente la suma total de la matriz. De esta manera se aprovecha la ejecución paralela, distribuyendo el trabajo entre varios hilos para reducir el tiempo total de cómputo.

.

Comparación entre ambos enfoques:

Al comparar ambos códigos, la diferencia principal está en el tiempo de ejecución. En la versión secuencial, el programa recorre bloque por bloque de forma lineal y tarda aproximadamente 0.0627 segundos. En cambio, la versión SMP con hilos utiliza ThreadPoolExecutor, lo que permite procesar los bloques en paralelo y reducir el tiempo total a unos 0.0215 segundos. Esto demuestra que el enfoque paralelo es más eficiente y veloz, ya que distribuye el trabajo entre varios hilos y lo ejecuta de manera simultánea, lo que resulta especialmente útil en tareas de mayor tamaño o complejidad. Por esta razón, se convierte en la opción más adecuada cuando se busca optimizar el rendimiento.

Ejercicio 2: Arquitectura SIMD (Single Instruction, Multiple Data).

Código secuencial:

El algoritmo naive corresponde al producto de matrices clásico mediante tres bucles anidados. Para cada posición (i, j) de la matriz resultado, se recorre toda la fila i de la primera matriz y toda la columna j de la segunda, acumulando el producto de cada par de elementos.

Este enfoque es totalmente secuencial: cada multiplicación y suma se realiza de una en una, sin ningún tipo de paralelismo explícito. Además, en Python cada operación conlleva la sobrecarga del intérprete, lo que ralentiza aún más el cálculo. Aunque es correcto y conceptualmente simple, presenta complejidad $O(n^3)$, lo cual hace que para tamaños grandes (por ejemplo 1000×1000) sea prácticamente inutilizable, tardando minutos u horas.

Básicamente, el código naive representa la forma más básica y secuencial de resolver el problema, útil como referencia, pero ineficiente en la práctica.

Código optimizado con NumPy (SIMD):

Por el contrario, el uso de NumPy con la operación A @ B delega la multiplicación a rutinas de bajo nivel escritas en C/Fortran, específicamente a BLAS (Basic Linear Algebra Subprograms).

Estas librerías están optimizadas para aprovechar:

- SIMD (Single Instruction, Multiple Data): una sola instrucción de CPU procesa varios datos en paralelo.

- Multithreading: divide el cálculo entre varios núcleos del procesador.

- Optimizaciones de memoria/cache: se trabaja por bloques de matrices para aprovechar mejor la jerarquía de memoria.

Gracias a esto, NumPy puede multiplicar matrices de 1000×1000 en pocos segundos, mostrando un rendimiento muy superior al enfoque naive.

Comparación del comportamiento:

Cuando se comparan ambos enfoques, se observa una diferencia de órdenes de magnitud en el tiempo de ejecución. Para tamaños pequeños (n=50 o n=100), el naive funciona correctamente pero ya es varias veces más lento que NumPy. A medida que el tamaño de la matriz crece, el tiempo del naive aumenta drásticamente de forma cúbica, mientras que NumPy escala mucho mejor gracias a la vectorización y paralelismo interno.

En cuanto a los resultados, ambos producen matrices equivalentes (hasta diferencias numéricas mínimas por la precisión de coma flotante). La diferencia no está en la exactitud, sino en la eficiencia computacional.

Conclusión:

La comparación evidencia que el algoritmo naive, al ejecutarse de forma completamente secuencial, no aprovecha las capacidades modernas del hardware. En contraste, NumPy utiliza rutinas optimizadas que explotan la arquitectura SIMD (Single Instruction, Multiple Data).

Las ventajas de SIMD se reflejan en la posibilidad de procesar múltiples datos en paralelo con una sola instrucción, reduciendo drásticamente el número de ciclos necesarios para completar el cálculo. Esto permite aprovechar mejor los registros vectoriales de la CPU, disminuir la latencia y aumentar el rendimiento global. Gracias a esta arquitectura, operaciones de gran escala como la multiplicación de matrices pueden ejecutarse en segundos en lugar de minutos u horas, logrando un desempeño notablemente superior al enfoque secuencial.

Tabla comparativa:

Aspecto	Naive (secuencial)	NumPy (SIMD/BLAS)
Estructura	Tres bucles anidados en Python	Llamada a librerías optimizadas en C/Fortran
Complejidad teórica	$O(n^3)$	$O(n^3)$ con optimizaciones internas
Paralelismo	No usa paralelismo, ejecución secuencial	Usa SIMD y multithreading

Tiempo en n=1000	Impráctico (muy lento)	Segundos (rápido y escalable)
Precisión de resultados	Correcto, mismo resultado que NumPy	Correcto, mismo resultado que naive
Uso práctico	Solo en matrices pequeñas o enseñanza	Útil y eficiente en aplicaciones reales

Ejercicio Integrador: Simulación de un sistema híbrido SMP-SIMD.

Descripción de las Estrategias:

Para este ítem se implementaron dos estrategias para el cálculo de la suma de los valores dentro de la matriz generada. La primera estrategia se desarrolló de forma secuencial, donde se recorrió la matriz dentro de ciclos for anidados para sumar de forma secuencial todos y cada uno de los valores que contiene. La otra estrategia, combina las arquitecturas SMP y SIMD para ofrecer una alternativa más eficiente donde al dividir la matriz en bloques y asignarle la suma de cada bloque a un hilo, se consigue distribuir el trabajo de tal forma que se aproveche mejor el hardware. La idea de hacerlo así, es debido a que se tiene la intención de comparar de manera clara cómo funcionan y cuánto tardan en ejecutarse, para ver también de forma sencilla la diferencia entre hacer todo en un solo flujo de instrucciones y repartir el trabajo en varios hilos que corren al mismo tiempo.

Inicialmente, se importaron las librerías necesarias para ejecutar el algoritmo. En ambos casos, se define una función generar_matriz(), que crea la matriz que contiene valores aleatorios entre 1 y 9 y que tiene dimensiones 10000×10000. Se usó una función para plantar una semilla y así hacer los resultados reproducibles en cuanto a la otra estragia.

En la estrategia secuencial se define una función suma_secuencial(), con la que se realiza el recorrido de la matriz y la posterior suma de los elementos contenidos en ella. También hay una función main() donde se llaman las otras funciones mencionadas y se ejecuta el código en general.

En la estrategia híbrida se definieron muchas más funciones, empezando con una llamada suma_bloque_filas() donde se suma con Numpy por filas los valores de un bloque de la matriz entregado como parámetro. La función generar_bloques() se encarga de retornar el bloque correspondiente de la matriz, de acuerdo a un índice que se le fue asignado. Se define la función funcion_target() donde se implementan las anteriores funciones y así establecer un flujo de trabajo para entregar a los hilos. Finalmente, se tiene la función main dónde ejecutar el código.

Dentro de la función principal main() en cada caso, se definieron las dimensiones de la matriz y con ellas se llamó a la función para generarla. En el caso secuencial, se inicia a contabilizar el tiempo de ejecución, se llama a la función para calcular la suma de forma secuencial y se detiene el conteo temporal. Los resultados de los tiempos de ejecución y el resultado final de la suma se imprimen en la consola. Por otro lado, en la estrategia híbrida, luego de definir las dimensiones de la matriz, también se definen las dimensiones de los bloques y su cantidad. También se crean dos listas, una para almacenar los resultados parciales de las sumas de los bloques y otra para almacenar los hilos.

Finalmente, se inicia el conteo de la ejecución, luego se crea un hilo por cada bloque, asignándole su actividad (La función target) de tal forma que se pueda ejecutar de forma paralela. Se inicia su ejecución y luego se espera a que todos los hilos hayan terminado, donde luego se suman con Numpy los resultados obtenidos en la lista de sumas parciales para así obtener la suma total. Se finaliza el conteo del tiempo de ejecución y se imprimen los resultados obtenidos.

Comparación de Resultados:

Al ejecutar ambas estrategias se obtuvieron valores relacionados a la suma total y al tiempo de ejecución. En el caso de la estrategia secuencial, se obtuvo:

```
juanm@Juan MINGW64 ~/OneDrive/Escritorio/UNIVERSIDAD/SÉPTIMO SEMESTRE/Infraestructuras Paralelas/Taller 4/taller4_infraestructur
as_paralelas (main)
$ C:/Users/juanm/AppData/Local/Programs/Python/Python312/python.exe "c:/Users/juanm/OneDrive/Escritorio/UNIVERSIDAD/SÉPTIMO SEME
STRE/Infraestructuras Paralelas/Taller 4/taller4_infraestructuras_paralelas/Ejercicio_3/ejercicio3_secuencial.py"
 Se inicia la suma de los elementos de una matriz 10000x10000


Suma total - Secuencial: 450008055
Tiempo - Secuencial: 5.7599 segundos
```

Donde se observa que demoró 5.76 segundos aproximadamente, en sumar todos los valores de la matriz de forma secuencial. Por otro lado, de la estrategia híbrida se obtuvo:

```
juanm@Juan MINGW64 ~/OneDrive/Escritorio/UNIVERSIDAD/SÉPTIMO SEMESTRE/Infraestructuras Paralelas/Taller 4/taller4_infraestructur
as_paralelas (main)
$ C:/Users/juanm/AppData/Local/Programs/Python/Python312/python.exe "c:/Users/juanm/OneDrive/Escritorio/UNIVERSIDAD/SÉPTIMO SEME
STRE/Infraestructuras Paralelas/Taller 4/taller4_infraestructuras_paralelas/Ejercicio_3/ejercicio3_SMP_SIMD.py"
 Se inicia la suma de los elementos de una matriz 10000x10000


Suma total - Híbrido (SMP - SIMD): 450008055
Tiempo - Híbrido (SMP - SIMD): 5.4941 segundos
```

De donde se tiene que su ejecución fue un poco menor, con un tiempo de 5.49 segundos aproximadamente. La diferencia de tiempos entre ambas estrategias es de 0.27 segundos, que aunque no parezca una gran diferencia, esto podría ser significativo cuando se tienen mucho más datos, representándose en ahorros importantes.

Si se calcula el speedup de la estrategia híbrida, con respecto a la estrategia secuencial, se obtiene un valor de 1.049, que indica que la versión híbrida fue casi un 5% más rápida que la versión secuencial. Lo anterior podría deberse a múltiples motivos, desde la implementación de la librería numpy para trabajar con arreglos y realizar las sumas de forma vectorial, hasta el uso de hilos para llevar a cabo los procesos de forma paralela.

Cosas a tener en Cuenta:

A pesar de que el uso de varios hilos; en teoría; debería ser más rápido que la versión secuencial, en la práctica esto no siempre ocurre, debido a que el rendimiento depende de varios factores, como es la estructura de datos usada, ya que al utilizar listas en python, cada hilo debe convertir luego los datos a arreglos Numpy antes de operarlos, consumiendo tiempo y anulando un poco el efecto de la paralelización.

Por otro lado, el crear demasiados hilos también puede resultar contraproducente, ya que podría conseguirse el efecto contrario al buscado al hacer que el sistema operativo incurra en planificación y conmutación del uso de estos. Por último, la creación de bloques requiere operaciones escritas en python puro, reduciendo la efectividad del paralelismo.

ACCESO AL REPOSITORIO.

Para acceder al repositorio de GitHub con los scripts desarrollados, haga click en el siguiente enlace:

Enlace al Repositorio de GitHub