🔍 Parte 1: Medir rendimiento de un código secuencial

**✅ Discusión:**

**💭 ¿Qué hace lento este código?**

1. **Iteración secuencial**:
   * El código **suma uno por uno** los valores del 0 hasta n-1. Eso implica que se hacen 10 millones de iteraciones.
2. **Acumulación secuencial**:
   * Cada iteración hace una operación de suma y una asignación a la variable total, lo cual genera un cuello de botella porque todo ocurre en un solo hilo (sin paralelismo).
3. **Uso intensivo de CPU**:
   * Es una operación completamente dependiente del procesador (CPU-bound), ya que está realizando muchas operaciones aritméticas sin esperar a entrada/salida.
4. **Acceso a memoria**:
   * total e i están en registros o memoria caché, lo cual es rápido, pero el **acceso repetido a estos valores** y su **actualización constante** todavía lleva tiempo, sobre todo si el valor crece y necesita más espacio.

**¿Cómo accede a memoria?**

* Se están usando **variables escalares** (total, i) que probablemente están en **registros** o **caché L1**.
* Aun así, **cada iteración necesita acceder a total para leerlo y luego escribir el nuevo valor**, lo que implica **lecturas y escrituras constantes a memoria**, aunque sea a nivel de caché.

**¿Qué pasaría si lo dividimos?**

Si dividimos el trabajo en **bloques paralelos**, por ejemplo con **threads** o **procesos** que trabajen sobre una porción del rango, podríamos:

* Aprovechar **múltiples núcleos del procesador**.
* **Reducir el tiempo total de ejecución**, porque varias partes del cálculo se harían en paralelo.
* Luego, solo tendríamos que **sumar los resultados parciales** al final.

Parte 2: Optimización usando operaciones vectorizadas (SIMD)

**Discusión:**

**❓ ¿Por qué NumPy es más rápido?**

NumPy es más rápido por varias razones fundamentales:

**⚙️ 1. Vectorización y SIMD**

* NumPy internamente usa librerías optimizadas en **lenguaje C**, que aprovechan instrucciones **SIMD (Single Instruction, Multiple Data)**.
* SIMD permite que una sola instrucción de la CPU procese **varios datos al mismo tiempo** (por ejemplo, sumar varios números a la vez), lo que es mucho más eficiente que hacerlo número por número como en un for.

**🔁 2. Menor cantidad de instrucciones de alto nivel**

* En el código original (suma\_normal), Python ejecuta **una instrucción por iteración**, lo que representa **millones de instrucciones**.
* NumPy reduce esto a unas **pocas llamadas optimizadas**, lo que implica **menos ciclos totales**, y por tanto, menor **CPI (Cycles Per Instruction)**.

**💾 3. Mejor uso de memoria y caché**

* NumPy almacena los datos de forma **contigua en memoria**, lo que mejora el acceso a memoria (menores fallos de caché).
* La CPU puede **predecir y precargar** los datos en caché con mayor eficacia.

**🧮 4. Optimización del backend (BLAS, LAPACK, MKL, etc.)**

* NumPy puede usar librerías como **Intel MKL** o **OpenBLAS**, que están altamente optimizadas para operaciones matemáticas en matrices y vectores.

**¿Qué ocurre a nivel de CPU?**

* Se aprovechan instrucciones SIMD para realizar operaciones en paralelo dentro de un solo núcleo.
* Se reducen los accesos individuales a variables y se trabaja con **vectores enteros de datos en registros SIMD** (por ejemplo, usando AVX o SSE en CPUs modernas).
* El CPI (ciclos por instrucción) es mucho menor, ya que **una sola instrucción puede hacer el trabajo de muchas**.

**✅ Comparación final:**

| **Enfoque** | **Tiempo esperado** | **Características clave** |
| --- | --- | --- |
| for en Python | Más lento | Código secuencial, sin paralelismo ni SIMD |
| multiprocessing | Más rápido | Aprovecha múltiples núcleos (paralelismo) |
| NumPy (vectorizado) | El más rápido | SIMD + código optimizado + menor CPI |

Parte 3: Procesamiento paralelo con múltiples núcleos

**Discusión:**

**¿Qué relación tiene esto con arquitecturas multinúcleo? ¿Cómo se distribuye la carga?**

**🧠 Relación directa con arquitecturas multinúcleo:**

Este ejemplo aprovecha el **paralelismo a nivel de proceso**, que es uno de los pilares fundamentales de las **arquitecturas de procesadores multinúcleo**. En lugar de que todo el trabajo lo haga un solo núcleo (como en el código secuencial), aquí se utiliza un **pool de procesos** donde cada uno puede correr en un **núcleo distinto del procesador**. Esto permite que **múltiples operaciones se realicen al mismo tiempo**, reduciendo el tiempo total de ejecución y aumentando la eficiencia general del sistema.

**🧩 Detalles clave sobre cómo se aprovecha la arquitectura multinúcleo:**

**1. Distribución de la carga (workload distribution):**

* El conjunto de datos (numeros = list(range(10\*\*6))) se **divide automáticamente** en bloques.
* Cada proceso hijo del Pool recibe un **subconjunto del total de elementos**.
* Esa subdivisión permite que **cada núcleo realice cálculos independientes**, como si fueran mini-programas corriendo en paralelo.

**2. Procesamiento concurrente real:**

* A diferencia del multithreading en Python (que está limitado por el Global Interpreter Lock o GIL), el uso de multiprocessing lanza **procesos independientes**, cada uno con su propio intérprete de Python y espacio de memoria.
* Esto permite **ejecución verdadera en paralelo** en varios núcleos físicos o lógicos (en arquitecturas con hyperthreading).

**3. Escalabilidad:**

* Este tipo de procesamiento se **escala eficientemente**: mientras más núcleos tenga el procesador, **mejor rendimiento se puede obtener** (hasta cierto punto, limitado por la sobrecarga de coordinación entre procesos).
* En procesadores modernos (por ejemplo, con 4, 8 o 16 núcleos), el tiempo de ejecución puede **disminuir hasta 4, 8 o más veces**, dependiendo del problema.

**💾 Acceso a memoria y eficiencia:**

* Cada proceso accede únicamente a su porción de datos, lo cual **minimiza la competencia** por recursos compartidos.
* Se reduce la latencia y mejora la localización en caché, especialmente si los datos se acceden secuencialmente.
* Esto reduce el **número total de ciclos por instrucción (CPI)** de forma efectiva, ya que **más trabajo se hace por ciclo** a nivel de sistema.

**🧬 Conexión con principios de arquitectura de computadores:**

| **Concepto** | **Aplicación en este ejemplo** |
| --- | --- |
| **Multinúcleo** | Cada proceso se ejecuta en un núcleo distinto. |
| **Paralelismo a nivel de datos** | Cada núcleo procesa un bloque diferente de números. |
| **CPI efectivo reducido** | Más operaciones en paralelo → menos tiempo total → menos ciclos por resultado. |
| **Localidad de datos** | Cada proceso trabaja con su propia copia de datos → menos fallos de caché. |
| **Overhead de sincronización** | Mínimo, ya que no hay comunicación entre procesos durante el cálculo. |

Parte 4: Estructuras condicionales dentro del bucle

**Discusión:**

**🔍 ¿Por qué la segunda versión es más lenta?**

La segunda versión (version\_2) es más lenta porque incluye una **estructura condicional dentro del bucle**, lo que tiene varias implicaciones a nivel de arquitectura:

**⚙️ 1. Evaluación redundante de la condición:**

* En version\_2, la condición if len(data) > 0: se evalúa **en cada una de las 10 millones de iteraciones**.
* En version\_1, se evalúa **una sola vez**, antes del bucle.
* Este cambio parece pequeño, pero introduce una instrucción extra en **cada ciclo**, multiplicando la carga de trabajo innecesariamente.

**🧠 2. Impacto en el CPI (Cycles Per Instruction):**

* Cada vez que se evalúa una condición, la CPU debe ejecutar instrucciones de **comparación y salto condicional**.
* En version\_2, esto significa **10 millones de saltos condicionales adicionales**.
* Esto **incrementa el CPI**, ya que se necesitan **más ciclos de reloj para completar el mismo trabajo útil** (la suma).

**🔁 3. Menor eficiencia de predicción de saltos:**

* Las CPU modernas usan **predictores de salto (branch prediction)** para anticipar el flujo del programa.
* Si bien la condición if len(data) > 0 es siempre verdadera, **su ubicación dentro del bucle impide que el predictor optimice completamente su ejecución**, provocando **fallos de predicción y penalizaciones de rendimiento**.

**💾 4. Acceso innecesario a memoria:**

* Aunque len(data) es constante, Python **evalúa la longitud de range(10\*\*7) cada vez**, y esto representa un **acceso repetido al objeto**.
* Esto implica llamadas constantes a funciones internas, creando más instrucciones que no aportan valor real.

**📊 Resumen comparativo:**

| **Versión** | **Evaluaciones de if** | **Total de instrucciones** | **CPI efectivo** | **Rendimiento esperado** |
| --- | --- | --- | --- | --- |
| version\_1 | 1 | Bajo | Menor | Más rápido |
| version\_2 | 10⁷ (10 millones) | Alto | Mayor | Más lento |

Parte 5: Orden de acceso a memoria (cache-friendly vs. no)

**Discusión:**

**🧠 ¿Por qué acceder por columnas es más lento?**

La respuesta se encuentra en **cómo se almacenan las matrices en memoria** y cómo las CPU modernas usan **caché** para acelerar el acceso a los datos.

**🗃️ 1. Organización en memoria (row-major order):**

* En **lenguajes como C, C++ y Python (NumPy)**, las matrices se almacenan en **orden fila a fila (row-major)**.
* Esto significa que los elementos de la **misma fila están contiguos en memoria**:  
  Por ejemplo, la fila i=0:  
  matriz[0][0], matriz[0][1], matriz[0][2], ..., matriz[0][999]  
  están uno al lado del otro en memoria.

**🧱 2. Caché de CPU y localidad espacial:**

* Las CPU modernas cargan los datos desde memoria principal a **caché en bloques** (por ejemplo, bloques de 64 bytes).
* Cuando accedes de forma secuencial a los datos (como en acceso\_filas()), se aprovecha la **localidad espacial**: una sola carga en caché trae múltiples valores útiles.
* En cambio, en acceso\_columnas(), se accede **salteando entre filas**:

python

CopiarEditar

matriz[0][0], matriz[1][0], matriz[2][0], ...

Estos **no están contiguos en memoria**, así que **cada acceso puede requerir una nueva carga en caché**, generando muchos **fallos de caché (cache misses)**.

**🧠 3. Impacto en el rendimiento y el CPI:**

* Los fallos de caché obligan a la CPU a esperar a que se traigan los datos desde la memoria principal (mucho más lenta).
* Esto **aumenta los ciclos por instrucción (CPI)** y ralentiza drásticamente el procesamiento.
* Por eso, acceso\_columnas() es **notablemente más lento**, aunque haga exactamente las mismas operaciones que acceso\_filas().

**🧬 Conexión con arquitectura de computadores:**

| **Concepto** | **Aplicación en el ejemplo** |
| --- | --- |
| **Row-major memory layout** | Optimiza el acceso por filas (datos contiguos) |
| **Cache locality** | El acceso por filas maximiza la **localidad espacial** |
| **Cache misses** | El acceso por columnas genera accesos no contiguos → más fallos |
| **Rendimiento (CPI)** | Mayor cantidad de fallos de caché → mayor CPI → menor rendimiento |