

Introducción Paralelos

Los sistemas paralelos lo encontramos en dispositivos de uso diario y en la nube

Programar en paralelo

Conocimiento del hardware del computador

Accesos de la CPU a la memoria

Hit + Miss

Tasa de acierto | Tasa de Fallo

$$\text{Hit Ratio} = \frac{\text{Hit}}{\text{Hit} + \text{Miss}}$$

Tiempo promedio de acceso a la memoria (cuando el acceso es simétrico)

$$T_{avg} = h \cdot T_c + (1-h) \cdot T_m$$

Ley de Amdahl: Límite inferior el cual una tarea por más elementos de procesamiento que tenga, no podrá reducir su tiempo de computo

$$\text{Latencia} = \frac{T}{(1-p) + \frac{p}{S}}$$

S = factor de paralelización

p = proporción de código paralelizable

Ley de Moore: Crecimiento del doble en el número de transistores en los procesadores cada 2 años

Venían incrementando (1973 - 2003) pero se han estabilizado en 3 GHz

Por un alto consumo de energía (Powerwall), discrepancias con la velocidad de CPU y memoria (Memory wall), límites al paralelismo de bajo nivel (ILP)

Para las nuevas arquitecturas de computo. No es útil la programación serial.

Patrones estructurados paralelos

Importa la SOLUCIÓN Y NO la implementación

Trampas seriales

Partes del Código que deben ser **seriales** o **secuenciales** pero no lo son

Valgrind

Se suele usar para la detección de errores en la gestión de memoria

Tool = CacheGrind

Brinda info de la cache

$$T_{avg} = h \cdot T_c + (1-h) \cdot (T_f + T_m)$$

Hay una tarea que tiene subtareas independientes entre si. El **span** de ese algoritmo se basa en el tiempo de aquella secuencia de instrucciones / **Tareas que toma más tiempo en ejecutarse**

Ejemplo: 30% de un programa está sujeto a **speedup** y la mejora en la ejecución es del doble ($S=2$)

Una tarea serial constituida de 4 partes consecutivas con porcentajes de tiempo de ejecución

$$P_1=0,11; P_2=0,18; P_3=0,23; P_4=0,48$$

$$S_1=1; S_2=5, S_3=20 \text{ Y } S_4=1,6$$

$$\text{Latencia} = P_1 + \frac{P_2}{S_2} + \frac{P_3}{S_3} + \frac{P_4}{S_4}$$

ILP wall: Hardware \Rightarrow paralelo, procesadores \Rightarrow paralelo inst. indp.

Técnicas para aprovechar la naturaleza paralela

Speculative execution:

Ejecutar instrucciones que se ejecutarán en el futuro

Pipelining: Dividir hasta en 10 etapas la ejecución de una instrucción

Patrones software paralelo:

Patrones nacen de estructurar "soluciones mejor conocidas" ofrecen soluciones de alto nivel y

Un programaador de alto nivel busca en el código partes paralelizables

Programas eficientes paralelos → reducir la cantidad de trabajo computacional

El **Objetivo** de parallelizar es minimizar su **Span** o tiempo de ejecución

reducir **Span** → reducir comunicación o acceso a los datos

Reducir tiempos de ejecución: Usar memoria compartida y la localidad → temporal espacial

Ejemplo 2: Un programa constituido por dos partes A y B donde $T_A = 3$ y $T_B = 1$

Si B se ejecutara 5 veces más rápido el speedup es:

$$\text{Latencia} = \frac{1}{1-0,25 + \frac{0,25}{5}}$$

Si A se ejecutara 2 veces más rápido el speedup es

$$\text{Latencia} = \frac{1}{1-0,75 + \frac{0,75}{2}}$$

Cosas importantes al parallelizar

- Cantidad total del trabajo computacional (hacer profiling)

El span

- Cantidad total de comunicación

Un vocabulario para facilitar la comunicación

Soluciones de alto nivel: Se enfocan en la algoritmia de la solución. El objetivo de **apps paralelas** es en el software escalable con muchos núcleos con poco modificar

Clase 10-9-24

Introducción al Perfilado

La optimización sirve para ser más rápido y consumir menos memoria.

Antes de optimizar, usar la técnica de **perfilado**, para saber si es necesario y en qué partes.

No es necesario optimizar si el código se ejecutara pocas veces, o cuando tome más tiempo optimizar que ejecutar.

Antes de optimizar se necesita que el código funcione sin errores, que este refactorizado y limpio, identificar partes innecesarias **perfilado**.

Es importante que el código tenga **claridad**.
EJ: `time.Perf_Counter()` y `time.Process_time()`

Un programa está **limitado** a la potencia de procesamiento disponible como la memoria, operaciones de entrada y salida, y latencia.

CPU-bound (tareas intensivas depende de la velocidad del procesador)

I/O-bound: tareas que esperan datos de un disco, base de datos o red.

Para medir muchos **fragmentos** de código sirve **Codetime**. Ya que envuelve estas funciones en clases, gestores de contexto y decoradores.

Los resultados serán los mismos en iguales condiciones.

Estos llamados recursivos pueden ser redundantes, para eso la **memorización**.

El perfilado sobrecarga el tiempo al registrar eventos. En algunos casos se pelea con esta **sobrecarga**.

Libreria Pyinstrument Perfilado **estadístico** captura estados en intervalos específicos.

Mantenibilidad Ya que aseguramos que funcione correctamente y que sea **entendible** para los demás y para mí.
Los **Cuellos de botella** pueden ser por comunicación de red u otros factores externos.

Perfilar identifica cuellos de botella, el principio de **Pareto** (80/20) optimizado

Con métricas se identifican los **puntos críticos** (cuellos de botella).

Los puntos críticos surgen por: uso excesivo de memoria, uso ineficiente de CPU, diseño de **datos subóptimo**.

Un algoritmo o estructura de datos también influye.

El módulo **time** es útil para verificaciones temporales en condiciones reales.

Para minimizar la influencia de **factores externos** se puede deshabilitar la basura de Python y repetir la medición.

El ruido y el tiempo de inicio afectan las mediciones.

Librería timeit

Evita errores comunes al medir el tiempo de fragmentos de código. Considera la carga del sistema, recolección de basura, y procesos concurrentes.

Es más preciso

Se usa en la consola o código.

Configuración timeit: Si el código necesita una configuración **timeit** lo hace.

cos, **Captura TODOS los llamados**.

Reduce Sobrecarga en comparación con un Perfilador determinista.

Ventanas Filtran llamadas que no afectan el rendimiento. La Sobrecarga es **uniforme y ajustable**. Dependiendo la tasa de muestreo, las funciones que retornan rápidamente

en el rendimiento

Para un **análisis dinámico** se ejecuta el código y recolectando datos del mundo real. Con pequeños volúmenes de datos.

Tipos de herramientas de perfilado en python

Tiempos: bibliotecas como "time" y "timeit" o "codetiming tercero".

Perfilador determinista: "profile", "CProfile", y "line_profiler".

Perfilador estadístico: "pyinstrument" y "Perf" (Linux). Medición del tiempo de ejecución con el módulo **time**.

La forma básica de **perfilado** en Python es con el temporizador **time**.

El **mejor resultado** es el que más se acerca más a la verdad.

Muchos tiempos = Ruido aleatorio

Limitaciones

Límites al recolectar **métricas** más detalladas para identificar **cuellos de botella**.

CProfile

Hay un Perfilado duro en Python (**Profile**) Y una extensión C

Buen rendimiento con menor sobrecarga. **Profile** se usa cuando no está disponible **cProfile**.

Ambos tienen Perfiladores **determinista** que rastrea todos los llamados a funciones cuantas veces lo hizo y **tiempo total**.

Pueden no aparecer en el reporte.

Se instala con **Pip**

No Puede manejar código con múltiples **hilos** o funciones en modulos de extensión en C

Para un análisis exhaustivo, usar **perf** en Linux

Perfilado Con Perf

Puede acceder a contadores de rendimiento de **Hardware** en algunas arquitecturas

Informa sobre toda la pila: Eventos de hardware, llamados a sistema, código de biblioteca. Etc.

Tiene una sobrecarga pequeña y ajustable (**eficiente**) para análisis

Solo está en linux y el código se ejecuta desde la fuente usando banderas de compilador para mejorar resultados

Perf analiza toda la app identificando cuales de botella en biblioteca o llamados.

Util para identificar Multithreading en Python

Capacidad de un procesador para ejecutar múltiples hilos de forma concurrente.

En CPU de un solo núcleo se logra con el cambio frecuente entre hilos **Cambio de contexto**: guarda el estado de un hilo y carga el estado del otro hilo

Para usarlo es así:
import threading

Para crear un nuevo hilo, se crea un objeto de la clase Thread la cual toma los parámetros target y args

Función a ejecutar

```
t1 = threading.Thread(target=funcion_objetivo, args=())
t2 = threading.Thread(target=otra_funcion, args=(args,))
```

Array: Arreglo ctype alojado en memoria compartida

Value: Objeto ctype alojado en memoria compartida

Los cambios en array y value por un proceso se ven reflejados en otros procesos

La Memoria Compartida
Sirve para sincronizar datos y evitar duplicación en cada proceso.

Server Process en Multiprocessing

Al iniciar un programa en PY se inicia un **Proceso Servidor**.

Problemas de rendimiento en hardware o S.O

Clase 18-9 Profiling Python

Procesos en Python

Un proceso es una instancia ejecutándose, el cual se compone de:

1. Un programa ejecutable
2. Datos del Programa (variables, espacios de trabajo, buffers, etc)
3. Contexto de ejecución
(estado del proceso)

Threading en Python

Un hilo es un **subconjunto de procesos**, unidad más pequeña de procesamiento. Entidad en un proceso programada para ejecución. Secuencia de instrucciones.

Componentes de un hilo dentro de un Bloque de

Para iniciar un hilo se usa el método **start()** t1.start()

Para esperar a que un hilo termine se usa el método **join()** t1.join()

Thread Pool

colección de hilos creados previamente y reutilizable

Concurrent.futures proporciona la clase ThreadPoolExecutor, facilita crear y gestionar el thread pool.

En el multithreading hay que tener cuidado con los **race condition** y **deadlocks**

Multiprocessing

Soportar más de un procesador. Aquí las apps se dividen en rutinas más pequeñas que se ejecutan independiente

Los procesos padres se conectan al servidor para crear nuevos procesos.

Un **Server Process** puede mantener objetos y permitir que otros procesos los manipulen usando proxies.

La clase **manager** controla un server process, permitiendo crear datos compartidos entre diferentes procesos.

Ventajas

- Soportan tipos de objetos

Un bloque de control de hilos (TCB)

1. Identificador de hilo: ID único de cada hilo **TID**

2. Punteros de pila: Apunta a la pila en el proceso. Contiene las variables locales

3. Contador de Programa

Registro de dirección de instrucción ejecutándose por un hilo

4. Estado del hilo: Ejecución, listo, en espera, iniciado, finalizado

5. Conjunto de registros del hilo: Registros asignados al hilo para cálculos.

6. Puntero del proceso padre: un puntero al Bloque de Control de Proceso (PCB) del proceso en el que vive el hilo

El SO asigna estos hilos a los procesadores, mejora rendimiento

Usar Multiprocessing

Las tareas comparten el mismo procesador, causando interrupciones y cambios de contexto. Por el otro lado se pueden ejecutar varias tareas al mismo tiempo, cada una en su propio procesador.

- Se ejecuta independiente
- Tiene su propio espacio en memoria

El módulo multiprocessing proporciona los objetos **Array** y **Value** para compartir datos entre procesos

arbitrarios (listas, diccionarios, colas, etc)

- Compartir procesos en diferentes computadoras por la red.

Desventajas

- MÁS LENTOS que usas objetos de memoria compartida

Comunicación entre procesos

El uso de múltiples procesos requiere comunicación entre ellos. Multiprocessing soporta dos canales de comunicación

Queue: Pasar mensajes entre procesos

Pipe: Intercambio de datos.

Queue: Comunica fácilmente los procesos en multiprocesamiento. Puede pasar objetos en PY.

Multiprocesamiento. Queue es similar a Queue. Queue

cola
Put inserta', get recupera

la **cola** coordina el trabajo entre múltiples procesos.

Pipe: Tiene solo dos puntos finales, recomendable para **comunicación bidireccional** entre 2 procesos.

Multiprocesamiento da la **Pipe()** retorna un par de objetos de conexión.

Cada objeto de conexión tiene métodos **Send()** y **recv()** para los mensajes

Pipe es más **eficiente** que Queue cuando solo se comunican dos procesos.

Clase 25-9 OpenMP

OpenMP es una API estándar para programación paralela en sistemas multiprocesador usada en C, C++, Fortran

Objetivo: Reducir tiempo distribuyendo tareas entre múltiples núcleos o procesadores.

Beneficio: Sencillo de implementar

Desafío: Que el programa sea eficiente requiere un esfuerzo adicional en la planificación de tareas.

Por qué usar OpenMP?

• En tareas simultáneas aprovecha múltiples procesadores. Aumenta eficiencia en cálculos complejos y repetitivos. Ideal para operaciones matemáticas paralelizable.

Operaciones Matemáticas en Paralelo Vector |

Facilmente paralelizables

Suma de dos vectores puede dividirse en sub-operaciones independientes

$$C_i = X_i + Y_i, \quad i=1, \dots, N$$

Cada suma se puede asignar a un procesador diferente sin dependencia entre datos.

Ventajas: Eficiencia máxima al no haber dependencias entre operaciones. Cada procesador hace su tarea independiente.

No fácil paralelizar

El producto punto de dos vectores, ya que es una secuencia de sumas que se acumulan en un solo valor escalar.

$$a = \sum_{i=1}^N X_i Y_i$$

Tiene resultados intermedios que dependen del orden de las sumas.

Reorganizar producto punto

Se divide en dos etapas

1) Calcular las multiplicaciones $X_i Y_i$ en paralelo.

2) Reducir (acumulando) las sumas

Es decir que cada procesador multiplica pares X_i, Y_i en paralelo. Para luego combinar los resultados usando **reducción**.

En la acumulación limita la eficiencia

Acceso a Memoria y Cache

Para trabajar eficientemente con el cache los datos deben estar almacenados de forma continua en la RAM

Importancia: el cache L1

Se optimiza si los datos están en ubicación **contigua**. Las variables declaradas en funciones se almacenan en el **Stack**, rápido acceso desde el cache. El cache lee la RAM en

bloques de **N bytes** contiguos ($N \approx 64$)

El acceso a localidades de memoria **distantes** afecta la eficiencia cache.

Lecturas Secuenciales VS No Contiguas

Acceso rápido: La lectura de memoria se hacen de forma secuencial hacia adelante (itt).

Acceso más lento: Lecturas hacia atrás (i-) o en patrones no secuenciales realentiza el acceso.

Hay que estructurar las operaciones de memoria para aprovechar la secuencialidad.

Paralelismo Real

Cada thread se ejecuta en un core

- El principal cuello de botella es el acceso a la RAM
- Existe un solo canal de comunicación (**bus de datos/direcciones**) con la RAM
- Si dos cores tratan de leer la RAM, una espera a que la otra termine.

El problema se agrava con más procesadores.

Cuello de Botella Y Cache L2

Para reducir latencia en el acceso a la RAM esta **L2**

En paralelo con memoria compartida complica el acceso eficiente:

- Si un core trata de actualizar la memoria que existe en el cache de otro core, se produce un **cache-fault**

- Uno de los cores accederá a L2 o RAM para sincronizarse (**costoso**)

- **Cache-hit:** Cuando un core accede a datos que ya están en su cache.

Arquitectura multicore

- Cada core tiene su cache L1
- Cache L2 compartido entre cores

Estrategias para maximizar el uso del cache

Cada core accede a su cache.

- Opera con variables en el **stack**
- Evitar guardar variables en **data**
- Reducir el número de escrituras al **heap**; trabajar en el **stack** y escribir resultados en el **heap**.
- Trabajar con bloques de memoria contiguos
- Asegurarse que cada core trabaje en localidades de memoria diferentes

Organización de funciones y memoria

Mas estrategias

- Las funciones que se usan juntas se almacenan cerca en **.lib**, **.a** o no en **.dll**, **.so**
- Minimizar dependencia entre cores para evitar conflictos de cache.

Optimizar acceso a cache → reduce tiempo de espera entre cores

NUMA (Non-Uniform Memory Access)

Computadores multicore donde cada procesador accede rápidamente a su banco de memoria.

- Acceder al banco de memoria ajeno es lento
- Necesita circuitos para la coherencia de cache de los cores

Mejora la **escalabilidad** de sistemas con múltiples procesadores

Trabajar con OpenMP

Esquema: facilita la escritura de código en paralelo.

Paraleliza con **memoria compartida**. Los procesadores acceden a la misma memoria.

Posix se basa en threads usando librería **libpthread** (Posix Threads).

OpenMP usa thread para ejecutar en paralelo, simplificando la creación de programas paralelos.

para compilar usa la bandera **-fopenmp**

Define número de threads

Por defecto es igual al número de cores del PC. Se puede establecer mediante variable de ambiente o en el código run-time

necesario incluir el header **omp.h**

Variables private y shared

OpenMP gestiona variables como

Shared: accesible y modificable por los threads

Private: copiadas al stack de cada thread

Control de iteraciones con Schedule

OpenMP tiene varios tipos de **Schedule** para distribuir iteraciones.

Static: Divide iteraciones en bloques de tamaño fijo

Dynamic: Asigna bloques dinámicos a cada thread

guided: como dynamic pero bloques más pequeños.

auto: decisión automática del compilador o SO

runtime: El schedule se determina en tiempo de ejecución.

Optimización de código compilado

- Habilitar SSE para operaciones matemáticas

Incrementar el nivel de optimización (-O3).

Desactivar chequeo de operaciones de punto flotante (-ffast-Math)

No compilar asserts del código (-DNDEBUG)

¿Cuáles son algunos de los puntos críticos que se pueden identificar con el perfilado de software?

Seleccione una o más de una:

- a. Uso excesivo de memoria ✓
- b. Uso ineficiente de la CPU ✓
- c. Diseño de datos subóptimo ✓
- d. Calidad del teclado
- e. Resolución de pantalla baja

Las respuestas correctas son: Uso excesivo de memoria, Uso ineficiente de la CPU, Diseño de datos subóptimo

¿Qué factores pueden limitar el rendimiento de un programa, según el documento?

Seleccione una o más de una:

- a. El sistema operativo
- b. Las operaciones de entrada/salida y la latencia del programa ✓
- c. La potencia de procesamiento disponible ✓
- d. La cantidad de memoria ✓
- e. La calidad de la pantalla

Las respuestas correctas son: La potencia de procesamiento disponible, La cantidad de memoria, Las operaciones de entrada/salida y la latencia del programa

¿Cuáles son algunas de las funciones de temporización mencionadas en el documento?

Seleccione una o más de una:

- a. string.replace()
- b. array.sort()
- c. time.perf_counter() ✓
- d. time.process_time() ✓
- e. os.path()

Las respuestas correctas son: time.perf_counter(), time.process_time()

¿Qué ventajas ofrece el uso de la librería pyinstrument según el documento?

Seleccione una o más de una:

- a. Mejora la calidad del código
- b. Reduce el consumo de memoria
- c. Filtra llamadas insignificantes que no afectan el rendimiento general ✓
- d. Incrementa la velocidad de ejecución
- e. Tiene una sobrecarga uniforme y ajustable ✓

Las respuestas correctas son: Tiene una sobrecarga uniforme y ajustable, Filtra llamadas insignificantes que no afectan el rendimiento general

¿Qué técnica puede ayudar a reducir la influencia de factores externos en las mediciones de tiempo?

Seleccione una o más de una:

- a. Repetir la medición varias veces ✓
- b. Aumentar la resolución de la pantalla
- c. Deshabilitar el recolector de basura de Python ✓
- d. Cambiar de IDE
- e. Usar sistemas operativos más nuevos

Las respuestas correctas son: Deshabilitar el recolector de basura de Python, Repetir la medición varias veces

¿Cuáles de los siguientes son componentes de un hilo en Python?

Seleccione una o más de una:

- a. Conexión a internet
- b. Estado del Hilo ✓
- c. Consola de comandos
- d. Identificador del Hilo (TID) ✓
- e. Puntero de la Pila ✓

Las respuestas correctas son: Identificador del Hilo (TID), Puntero de la Pila, Estado del Hilo

¿Qué es un server process en multiprocessing?

Seleccione una o más de una:

- a. Un proceso que puede mantener objetos y permitir que otros procesos los manipulen usando proxies
- b. Un proceso encargado de gestionar los hilos
- c. Un proceso que se ejecuta en segundo plano
- d. Es controlado mediante la clase Manager ✓
- e. Un proceso que almacena datos en la nube

¿Cuáles son algunas características de un hilo (thread) en Python?

Seleccione una o más de una:

- a. Necesita su propio espacio de memoria separado
- b. Puede ejecutarse independientemente de otros códigos dentro de un programa ✓
- c. Requiere un sistema operativo de 64 bits
- d. Forma parte de un proceso ✗
- e. Tiene un identificador único (TID) ✓

Las respuestas correctas son: Puede ejecutarse independientemente de otros códigos dentro de un programa, Tiene un identificador único (TID)

¿Qué método se usa para iniciar un proceso en multiprocessing?

Seleccione una o más de una:

- a. begin()
- b. start() ✓
- c. activate()
- d. create()
- e. execute()

La respuesta correcta es: start()

¿Qué elementos se pueden compartir entre procesos usando el módulo multiprocessing en Python?

Seleccione una o más de una:

- a. Librerías de terceros
- b. Variables locales
- c. Colas (Queue) y Pipes ✓
- d. Arrays y Values ✓
- e. Módulos externos

Las respuestas correctas son: Arrays y Values, Colas (Queue) y Pipes

¿Cuáles son las etapas para parallelizar el producto punto de dos vectores?

Seleccione una o más de una:

- a. Calcular las multiplicaciones en paralelo ✓
- b. Imprimir los resultados en paralelo
- c. Crear un buffer de resultados intermedios
- d. Modificar el tamaño del cache
- e. Reducir los resultados acumulando las sumas ✓

Las respuestas correctas son: Calcular las multiplicaciones en paralelo, Reducir los resultados acumulando las sumas

¿Qué método se usa para iniciar un proceso en multiprocessing?

Seleccione una o más de una:

- a. begin()
- b. start() ✓
- c. activate()
- d. create()
- e. execute()

La respuesta correcta es: start()

¿Qué elementos se pueden compartir entre procesos usando el módulo multiprocessing en Python?

Seleccione una o más de una:

- a. Librerías de terceros
- b. Variables locales
- c. Colas (Queue) y Pipes ✓
- d. Arrays y Values ✓
- e. Módulos externos

Las respuestas correctas son: Arrays y Values, Colas (Queue) y Pipes

¿Cuáles son las ventajas del multithreading en Python?

Seleccione una o más de una:

- a. Se ejecuta más rápido en sistemas operativos antiguos
- b. Mejora la eficiencia en programas intensivos en CPU
- c. Requiere más memoria que un solo proceso
- d. Permite ejecutar múltiples hilos de forma concurrente ✓
- e. Facilita el acceso a redes externas ✗

Las respuestas correctas son: Permite ejecutar múltiples hilos de forma concurrente, Mejora la eficiencia en programas intensivos en CPU

¿Cuáles de los siguientes métodos se utilizan para trabajar con hilos en Python?

Seleccione una o más de una:

- a. start() ✓
- b. Thread() ✓
- c. join() ✓
- d. terminate()
- e. open()

Las respuestas correctas son: start(), join(), Thread()

¿Cuál es la directiva de OpenMP que permite parallelizar un ciclo for?

Seleccione una o más de una:

- a. init_thread()
- b. #pragma omp private
- c. #include <omp.h>
- d. #pragma omp parallel for ✓
- e. void parallel()

La respuesta correcta es: #pragma omp parallel for

¿Qué técnica se puede usar para parallelizar la multiplicación matriz-vector en OpenMP?

Seleccione una o más de una:

- a. Utilizar variables privadas para todos los datos
- b. Dividir las filas de la matriz entre los threads ✓
- c. Ejecutar el ciclo de forma secuencial
- d. Usar una memoria diferente para cada vector
- e. Cada thread escribe en su propia posición de salida ✓

Las respuestas correctas son: Cada thread escribe en su propia posición de salida, Dividir las filas de la matriz entre los threads

¿Cuál es la función de omp_get_thread_num() en OpenMP?

Seleccione una o más de una:

- a. Define el número de threads disponibles
- b. Establece la cantidad de memoria compartida
- c. Controla el acceso al cache
- d. Devuelve el número del thread actual ✓
- e. Devuelve el número de iteraciones por thread

La respuesta correcta es: Devuelve el número del thread actual

¿Cuáles estrategias pueden maximizar el uso del cache en sistemas multi-core?

Seleccione una o más de una:

- a. Asegurarse de que cada core trabaje en localidades de memoria diferentes ✓
- b. Operar con variables en el stack ✓
- c. Utilizar menos memoria RAM
- d. Evitar la paralelización
- e. Usar procesadores de un solo núcleo

Las respuestas correctas son: Operar con variables en el stack, Asegurarse de que cada core trabaje en localidades de memoria diferentes

¿Cuáles son los beneficios de optimizar el acceso a memoria con cache?

Seleccione una o más de una:

- a. El programa se vuelve más difícil de depurar
- b. Aumenta el tiempo de ejecución
- c. El cache L1 se optimiza cuando los datos están en ubicaciones contiguas ✓
- d. Mejora la eficiencia del acceso a la memoria ✓
- e. Disminuye la necesidad de paralelismo

Las respuestas correctas son: El cache L1 se optimiza cuando los datos están en ubicaciones contiguas, Mejora la eficiencia del acceso a la memoria

¿Cuáles son algunas herramientas de perfilado determinista mencionadas?

Seleccione una o más de una:

- a. matplotlib
- b. timeit
- c. cProfile ✓
- d. seaborn
- e. profile ✓

Las respuestas correctas son: cProfile, profile

¿Qué es el pipelining según el texto?

Seleccione una o más de una:

- a. Una estrategia para minimizar el consumo de energía
- b. Una forma de incrementar la velocidad del reloj
- c. Una técnica para parallelizar dividiendo la ejecución en etapas ✓
- d. Una técnica de sobrecarga de procesos
- e. Un método de aprovechar la naturaleza paralela del hardware

Las respuestas correctas son: Una técnica para parallelizar dividiendo la ejecución en etapas, Un método de aprovechar la naturaleza paralela del hardware

¿Cuáles son ejemplos de aplicaciones donde se encuentran sistemas de cómputo paralelos?

¿Qué ventajas ofrece el uso de un Thread Pool en Python?

Seleccione una o más de una:

- a. Teléfonos celulares ✓
- b. Consolas de videojuegos
- c. Computadoras cuánticas
- d. Laptops ✓
- e. Supercomputadoras

Las respuestas correctas son: Teléfonos celulares, Laptops

Seleccione una o más de una:

- a. Mejorar la gestión y el rendimiento de la concurrencia ✓
- b. Facilitar la comunicación entre hilos y procesos
- c. Aumentar el número máximo de hilos permitidos por el sistema operativo
- d. Ejecutar tareas en un entorno distribuido
- e. Reutilizar hilos para ejecutar múltiples tareas ✓

Las respuestas correctas son: Reutilizar hilos para ejecutar múltiples tareas, Mejorar la gestión y el rendimiento de la concurrencia

¿Cuál es la conclusión respecto a los "walls" en el diseño de procesadores?

Seleccione una o más de una:

- a. El paralelismo interno del hardware ha llegado a sus límites ✓
- b. La frecuencia del reloj seguirá aumentando
- c. Los programadores deben escribir programas explícitamente paralelos ✓
- d. El consumo de energía no es un problema relevante
- e. La memoria caché ha eliminado todos los cuellos de botella

Las respuestas correctas son: El paralelismo interno del hardware ha llegado a sus límites, Los programadores deben escribir programas explícitamente paralelos

¿Cuál es el objetivo principal de la parallelización según el texto?

Seleccione una o más de una:

- a. Aumentar la frecuencia del reloj
- b. Reducir el span ✓
- c. Reducir la cantidad de transistores
- d. Minimizar el tiempo de ejecución ✓
- e. Mejorar la eficiencia energética

Las respuestas correctas son: Reducir el span, Minimizar el tiempo de ejecución

¿Cuáles son los temas abordados en la introducción a la programación en paralelo?

Seleccione una o más de una:

- a. Procesamiento en paralelo con GPUs
- b. Redes neuronales profundas
- c. Patrones de programación en paralelo ✓
- d. Ley de Amdahl y ley de Moore ✓
- e. Motivación a la programación en paralelo ✓

Las respuestas correctas son: Motivación a la programación en paralelo, Ley de Amdahl y ley de Moore, Patrones de programación en paralelo

¿Qué se debe tener en cuenta a la hora de parallelizar un programa?

Seleccione una o más de una:

- a. El número de ciclos for en el código
- b. El span del algoritmo ✓
- c. La cantidad total de comunicación ✓
- d. El tamaño del programa
- e. La cantidad total de trabajo computacional ✓

Las respuestas correctas son: La cantidad total de trabajo computacional, El span del algoritmo, La cantidad total de comunicación