Clase 02: Introducción a la paralelización

Introducción a la paralelización

¿Para que programar en paralelo?

1. Tenemos sistemas multinucleo, que nos permite en TIEMPO REAL tener tareas paralelas
2. Cuando programamos lo hacemos de forma secuencial 1,2,3,4,...n, En ciertas situaciones las instrucciones son INDEPENDIENTES ¿Porque no aprovechar que tenemos un sistema multinúcleo?
3. Paralelización objetivo: Reducir el tiempo de ejecución al partir tareas independientes
4. Ley Amdahl hasta que punto puedo paralelizar, hasta la tarea secuencial que no se puede partir más grande

```
int arr[] = new int[100000]
for (int i = 0; i<=100000, i
    arr[i] = i;
}

for (int i = 1; i<=100000, i
    arr[i] = arr[i-1]+arr[i]
}
```

5. Ley moore: Numero de transistores por año cumplio bien hasta principios de los 2000

6. Otros aspectos: Frecuencia (generación de la señal y eficiencia energética), aspectos físicos aislamiento
7. Cuello de botella: Persistente → Ram → Cache → Registros (CPU)

Sistemas paralelos

1. Las aplicación en la nube suelen ser naturalmente paralelas
2. Sistemas de hardware son limitados, no puede crecer a necesidad (estamos limitados por el número de hilos que puede atender un sistema)
3. Cloud: Los sistemas de hardware son elásticos (es decir pueden cambiar a necesidad)
4. Determinismo: En paralelización las tareas se ejecutan en cualquier orden, es necesario garantizar que todas las tareas se ejecuten y den el mismo resultados

```
lst = [1,2,3,4]
#ls = [1,3,6,10]

for i in range(1,4):
    lst[i] += lst[i-1]

[1,4,6,10]

[1,3,5,9]
[1,2,6,7]
```

Programación en paralelo	• Trabajamos de forma secuencial, no se aprovecha la capacidad multinucleo de los sistemas
	• Programación en paralelo (paradigma)
	  ◦ Partición de tareas: Mapeo: División de tareas
	  ◦ Unificación de las tareas reduce la combinación de los resultados
	  ◦ Estructuras para garantizar determinismo

Paralelización C++

Recursos

1. Instalar en Windows https://www.mingw-w64.org en Linux/MAC build-essential/gcc
2. Visual Studio Code:
   - **C/C++ Microsoft**
   - **C/C++ Extension Pack Microsoft**

Librerías

1. Archlinux https://archlinux.org/packages/extra/x86_64/onetbb/ y https://archlinux.org/packages/extra/x86_64/clthreads/
- Debian/Ubuntu https://packages.debian.org/source/sid/onetbb https://packages.debian.org/buster/libboost-thread-dev
- Windows https://www.intel.com/content/www/us/en/docs/onetbb/get-started-guide/2021-6/install-onetbb-on-windows-os.html y https://github.com/GerHobbelt/pthread-win32

Paralelización C++

thread Lanzar hilos con un proceso, debemos partir este proceso por ende es necesario utilizar variables de inicio y fin, nosotros definimos cómo lo manejamos.

```
gcc -o exe archivo.cpp -lthread
```

bb Lanzar hilos con respecto a una colección for o reduce (datos) o tareas, la librería se encarga de la gestión de hilos, variable de inicio y final de recorrido de la colección.

```
gcc -o exe archivo.cpp -ltbb
```

https://gist.github.com/cardel/8fce3ebe2eff97479877d3083471217b

Clase 03: Introducción paralelización II (Ejercicio)

Instrucciones

En la terminal de codespaces escribir

```
sudo -s
apt-get update
apt-get install libtbb-dev libboost-thread-dev
```

hilos.cpp

```
#include <iostream>
#include <thread>
#include <vector>
#include <chrono>

using namespace std;
using namespace std::chrono;

const int VECTOR_SIZE = 1000000;
vector<long> v(VECTOR_SIZE);

void fillVector(int start, int end) {
    for (int i = start; i < end; i++) {
        v[i] = 10;
    }
}

void sumVector(int start, int end, long &result) {
    for (int i = start; i < end; i++) {
        result += v[i];
    }
}
```

```
int main() {
    auto start = high_resolution_clock::now();

    thread t1(fillVector, 0, VECTOR_SIZE / 2);
    thread t2(fillVector, VECTOR_SIZE / 2, VECTOR_SIZE);

    t1.join();
    t2.join();

    long result1 = 0, result2 = 0;
    thread t3(sumVector, 0, VECTOR_SIZE / 2, ref(result1));
    thread t4(sumVector, VECTOR_SIZE / 2, VECTOR_SIZE, ref(re

    t3.join();
    t4.join();

    auto stop = high_resolution_clock::now();
    auto duration = duration_cast<milliseconds>(stop - start)
    cout << "Time 2 threads: " << duration.count() << " ms" <<

    cout << "Result: " << result1 + result2 << endl;

    auto start2 = high_resolution_clock::now();
    fillVector(0, VECTOR_SIZE);
    long result = 0;
    sumVector(0, VECTOR_SIZE, result);
    auto stop2 = high_resolution_clock::now();
    auto duration2 = duration_cast<milliseconds>(stop2 - star
    cout << "Time seq: " << duration2.count() << " ms" << end
    cout << "Result: " << result << endl;

    return 0;
}
```

```
g++ -o hilos hilos.cpp -lpthread
./hilos
```

hilostbb.cpp

```
#include <iostream>
#include <tbb/parallel_for.h>
#include <tbb/blocked_range.h>
#include <tbb/parallel_reduce.h>
#include <vector>
#include <chrono>

using namespace std;
using namespace std::chrono;

const int VECTOR_SIZE = 1000000;

vector<long> v(VECTOR_SIZE);

void fillVector(){
    tbb::parallel_for(
        tbb::blocked_range<int>(0, VECTOR_SIZE),
         [&](tbb::blocked_range<int> r){
        for (int i = r.begin(); i < r.end(); i++) {
            v[i] = 10;
        }
    });
}

void sumVector(long &result) {
    result = tbb::parallel_reduce(
        tbb::blocked_range<int>(0, VECTOR_SIZE),
        0,
        [&](tbb::blocked_range<int> r, long init) -> long {
            for (int i = r.begin(); i < r.end(); i++) {
                init += v[i];
            }
            return init;
        },
        [](long x, long y) -> long {
            return x + y;
```

```
        });
}

int main() {
    auto start = high_resolution_clock::now();

    fillVector();

    long result = 0;
    sumVector(result);

    auto stop = high_resolution_clock::now();
    auto duration = duration_cast<milliseconds>(stop - start)
    cout << "Time TBB: " << duration.count() << " ms" << endl

    cout << "Result: " << result << endl;

    return 0;
}
```

```
g++ -o tbb hilostbb.cpp -ltbb
./tbb
```

Consigna

Calcular el angulo entre dos vector u y b

1. Llenar u y v con numeros aleatorios entre 1 y 100
2. Calcular u.v
    a. Calcula la multiplicación punto a punto de u y v, eso lo guarda en otro vector
    b. Suma los elementos del vector generado
3. Calculan la norma de u y v, la norma la raiz de la suma de los cuadrados
4. Toman el u.v y la multiplicación de las normas, dividen ambos resultados
5. Sacar el arcocoseno, recordar que lo necesito en grados

Clase 04: Paralelización II

Limitaciones

- Trampa serial: No se puede paraelizar suponiendo que las tareas van a tardar 1/n veces (n hilos), por que hay tareas que no se pueden paralelizar
- Recursos compartidos
    - Memoria: localidad temporal: Acceso a las posiciones de memoria compartidas por hilos
    - Memoria: localidad espacial: Cantidad de memoria gestionada por cada hilo (secuencialidad de los datos)
- Herramientas de profiling: Nos permiten estudiar cómo se comporta en memoria y en ejecución los programas

---

Rendimiento de la cache

- h la tasa de acierto (HitRatio)
- Tm Tiempo de acceso memoria RAM
- Tc Tiempo de acceso memoria cache

Acceso paralelo

$$T_{avg} = hT_c + (1-h)T_m$$

Acceso jeraquico/Secuencia

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

Dependencias	• Numeros de procesadores disponibles
	• Dependencias de tareas
	• Ley de Amdahl
	  ◦ p es la parte del código que se puede paralelizar
	  ◦ s es el número de hilos que puedo lanzar
	  ◦ T es el tiempo secuencial
	$$L = \frac{T}{(1-p) + \frac{p}{s}}$$

Elementos a tener en cuenta	• La cantidad total de trabajo: Para resolver una tarea al menos vamos a tener que realizar TODAS las operaciones que la involucran (profiling) • El span: El tiempo de la tarea secuencial que se demora más • La cantidad total de comunicación (Gestión de hilos y memoria en el programa)
Limitaciones con respecto al crecimiento de la capacidad	1. Ley de moore: Cada dos años tenemos el doble de transitores (propuesta en los años 70) pero que se cumplio hasta los años 2000 2. Tasas de reloj se estabilizaron (no tienen un crecimiento significativo) 3GHz

Tasa de reloj	1. Power all: Entre más frecuencia se consume más energia (relación es exponencial) es necesario usar refrigeración especial cuando un procesador se overclockea 2. Memory wall: La memoria y la CPU no tienen la misma frecuencia 3. Restricciones de bajo nivel sobre paralelismo
Estrategias para paralelizar	• Formas: Paralelización de tareas y de datos • Pipeline: Ejecutar las tareas en paralelo • Ejecución especulativa: Adelantar ejecuciones no dependientes (hay que predecir)

Resumen

- La velocidad de algoritmo no depende directamente la tasa de reloj (hay limitaciones)
- El paralelismo por hardware está en sus limites
- Debe reducirse el acceso a memoria
- Los procesadores tienen instrucciones de paralelización pero el Kernel debe de proveerlas para aprovecharlas

Patrones de software en paralelismo	1. Son a base de la experiencia (carecen de un marco teorico que los respalde 100%)

2. Nos permiten construir aplicaciones de alto nivel para problemas complejos nos preocupamos en cómo se paraleliza ni por el hardware, si no lo que queremos paralelizar
3. Muchos patrones. Pipeline (tareas), map→reduce (datos)
4. Tener en cuenta las dependencias entre los datos y las operaciones (caso de addme)

Clase 05 Profiling

¿Que es profiling?

- Identificar cuello de botella (Instrucciones que lo hacen lento) y que potencialmente se pueden mejorar
- Consumo memoria, CPU, recursos I/O
- El objetivo es buscar areas de posible mejora

Pasos para optimizar

1. Ejecutar pruebas: Nos ayudan a encontrar los cuellos de botella
2. Refactorización
3. Perfilado: Identificar las partes del código que dan mayor problema en la ejecución

80% problemas esta dado por el 20% del codigo

Herramientas

1. Temporizadores: Toman tiempos, proceso o del sistema operativo
2. Perfiladores deterministicos: La ejecución del programa
3. Perfiladores estocasticos: Toman una parte y estima el tiempo total.

Ambientes virtuales en Python

Linux

```
virtualenv venv

source venv/bin/activate
pip list
pip install numpy
```

WIndows

Hacer en Git

```
python -m pip install virtualenv ;;Gente terca en Windows
python -m virtualenv venv
source venv/Scripts/activate
```

Perf

Es una utilidad de Linux para perfilar programas (ejecuando) o que esten cargado (requerimos permisos de root)

```
perf record <ejecuacion>
perf report
```

Clase 06 Hilos y procesos en Python

Multihreading

¿Que es?

El hilo es la unidad fundamental de un proceso que se puede ser ejecutada por la CPU

Es una secuencia de instrucciones

Un hilo es un subconjunto de un proceso

Partes de un hilo

1. ID Identificador
2. Puntero de pila: Sección de la pila que pertenece al hilo (Contexto)
3. Contador de programa: Ubicación en memoria de la siguiente instrucción a ejecutar
4. Estado del hilo: running, ready, wait, initialize, closed
5. Conjunto de registro del hilo (Data en el procesador)
6. Puntero al proceso padre (Un puntero al proceso padre (donde fue lanzado) del hilo)

Multhreading

- Ejecutar varios hilos dentro de un CPU
- Cuando un procesador varia entre hilos debe un hacer un cambio de contexto (Valores en los registros)

Thread pool	1. Es una colección de hilos para ser lanzandos de forma recurrente 2. Contamos con un Threadpoolexceutor para lanzar los trabajos 3. Este se encarga de gestionar los hilos

Multiprocessing

¿Que es?	Es la capacidad de un sistemas de tener más de un procesador Las aplicaciones se pueden dividir en rutinas más pequelas que se ejecutan en procesos independientes • Multiprocesador: Una board con más de un CPU • MUltinucleo: Varios nucleos dentro de la mism CPU
Datos entre los procesos	• Cada proceso tiene su propio CONTEXTO • Un contexto es el conjunto de variables
Memoria compartida	• Las librerías nos ofrecen colecciones de datos sincronizadas • Estas colecciones son COMPARTIDAS entre procesos

Servidor de procesos

¿Que es un servidor de procesos?	1. Cuando inicia python inicia un servidor de procesos
	2. Este se encarga de lanzar los otros procesos
	3. El modulo multiprocessing propociona Manager nos permite controlar los procesos
	4. Ventajas:
	    a. Soporta tipos de datos
	    b. Puede ser compartidas entre procesos
	5. Desventajas:
	    a. Latencia (son más lentos)

Pipe	Sockets para el intercambio de información (de cualquier tipo)
Queue	Cola de objetos

Clase 07 OpenMP

¿Que es?	API: Application Programming Interface Enfocada a sistemas multinucleo Eficiente para paralelizar diferentes procesos Es ampliamente configurable e incluye directivas para paralelización de tareas, iteradores, ciclos, entre otros Nos permite controlar como iteramos Sin embargo, en este curso sólo abordaremos los **fundamentos**

Paralelización	• Tareas faciles de parelelizar: Suma de vectores, operaciones **independientes**
	• Tareas dificiles de pararelizar: Interoperables (producto punto)
	  ◦ Multiplicación 1 a 1 w[i] = v[i]*u[i]
	  ◦ Suma de los resultados $\sum_{i=0}^{n} w[i]$

Acceso a la memoria Cache

Limitaciones	El acceso esta optimizado para operaciones secuenciales en memoria, el acceso aleatorio o salteado produce latencias

cache fault	Es cuando un core intenta actualizar la memoria cache utilizada por otro core La solución es utilizar diferentes niveles L1 (que cada core una independiente) L2 y L3 que son compartidos
NUMA	SIstema multicore (un solo procesador multiples nucleos), cada procesador tiene su propia memoria cache

OpenMP

Paralelización basada en hilos

1. Deben instalar openmp / libopenmp
2. Vamos a utilizar unos bloques para indicar que queremos para parelizar

```
#pragma omp <instrucciones>
{

}
```

```
g++ -o prog archivo.cpp -fop
```

Gestión variables	• Privada: Para cada hilo • COmpartida: Compartida entre todos los hilos
Scheduling	Gestión de hilos

	• Static: lanza los hilos en bloques de tamaño fijo • Dynamic: Asigna bloques dinamicamente • guided: Es lo mismo que dynamic pero los bloques mas pequeños • auto: Lo selecciona el compilador o S.O • runtime: Tiempo de ejecución
Evitar condiciones de carrera	Condición critical que fuerza a que una instrucción sea ejecuta por un sólo hilo

Clase 08: Repaso

Acuerdos

Parcial 1 13 de Nov

Introducción al paralelismo

Motivación

$$\forall i \in S, c[i] = a[i] + b[i]$$

a = [1,2,3,4,5,6]

b = [2,4,6,8,10,12]

c = [3, 6,9,12,15,18]

Independientes

Aprovechar la capacidad de computo multinucleo para hacer tareas más rapido

---

Limitaciones

v = [1,2,3,4,5,6]

v[i] = v[i]+v[i-1]

v = [1,3,6,10,15,21]

1,2,3,4,5,6

2,4,1,3,6,5

v = [1,3,6,7,12,11]

Dependencia, este es un de los problemas del paralelismo

Dado el tamaño de la memoria cache no podemos saturarla con datos de procesos concurrentes

Dado el número de procesadores que no es infinito no podemos tener un número ilimitado de procesos

	• Localidad temporal • Localidad especial
Ley ahmdal	Esta ley nos dice que podemos tener una ganancia en tiempo en la ejecución de programas 1. Programas tienen partes secuenciales que no se pueden paralelizar 2. Supone que la capacidad de paralelizar es ilimitada (no es real) Nos da un indicio de cómo podemos obtener ganancia en tiempo paralelizando

Recursos

```
#Visualizar esquema del procesador
lstopo

#Ver tasas de fallo y escritura a la cache
perf stat -B dd if=/dev/zero of=/dev/null count=1000000
```

Profiling

¿Que es?	• Medir recursos de uso de una aplicación
	  ◦ Tiempo de ejecución
	  ◦ Uso de memoria
	  ◦ uso de Disco

- Operaciones en memoria cache
- etc

Librerias	• time • timeit • pyinstrument • cprofile • perf: Es una utilidad de Linux • hpy

Multithreading vs Multiprocessing

¿Cual es la diferencia?	• Un proceso esta compuesto por uno o más hilos • Cada proceso de INDEPENDIENTE en memoria, los hilos comparten la memoria del proceso • Podemos tener procesos en diferentes de nodos de cómputo