



Computación Paralela y Distribuida

2022-II

José Fiestas

05/09/22

Universidad de Ingeniería y Tecnología
jfiestas@utec.edu.pe

Unidad 3. Descomposición en paralelo

Objetivos:

1. Paralelismo directo
2. Uso de random en paralelo
3. Particionamiento, Divide y Vencerás

Paralelismo directo

Paralelismo directo: Procesamiento de imágenes

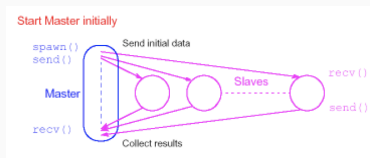
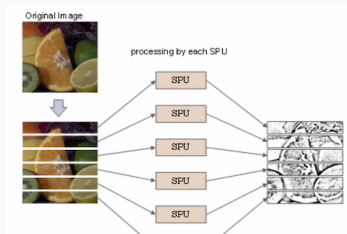
Donde

las tareas son completamente separables. Incluso no es necesaria la comunicación entre procesos

Ejemplo:

procesamiento de imágenes

Solo las coordenadas del pixel son modificadas, independientemente de los pixels vecinos (desplazamiento, escala, rotación, etc)



Se toma especial cuidado en la distribución de la imagen entre procesos, normalmente se separa la imagen en regiones cuadradas, o en líneas y columnas.

Paralelismo directo: Procesamiento de imágenes

Si se necesitan dos pasos de transformación para cada pixel: $(x' = x + \Delta x; y' = y + \Delta y)$

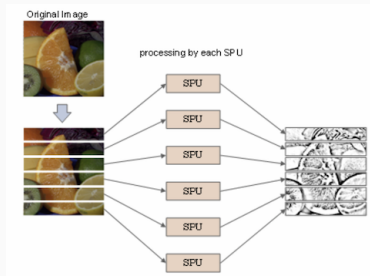
Tendremos una secuencia de cálculos de $n \times n$ pixels en tiempo $t_s = 2n^2$, es decir $O(n^2)$

Para p procesos, cada uno con 4 resultados (antiguas y nuevas coordenadas), es el tiempo de comunicación:

$$t_{comm} = p(t_{startup} + nt_{data})$$

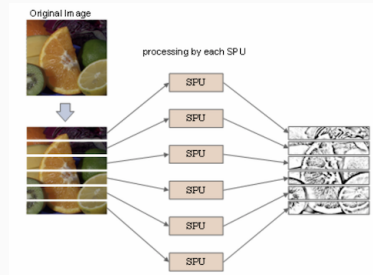
$$\text{Y el tiempo de cómputo: } t_{comp} = 2\frac{n^2}{p} = O(\frac{n^2}{p})$$

$$\text{Por consiguiente: } t_p = t_{comm} + t_{comp} = O(n^2/p + p + np)$$



Paralelismo directo: Procesamiento de imágenes

Asimismo,
el ratio cómputo/comunicación
es constante, $O(\frac{n^2/p}{p+np})$ si p es
constante. Ya que la comunicación
a los procesos y el retorno
es lo que más cuesta, arquitectura
de memoria compartida es óptima
para este tipo de problemas.



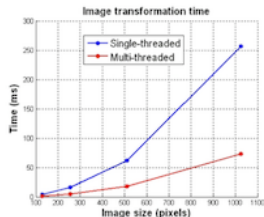
Paralelismo directo: Procesamiento de imágenes

Ejemplo: **Image wrapping**:

Transformación de giro de una imagen con respecto a un centro de rotación (c_x, c_y)

$$x' = (x - c_x)\cos\theta + (y - c_y)\sin\theta + c_x$$

$$y' = (y - c_y)\sin\theta + (y - c_y)\cos\theta + c_y$$



Ejemplo: Image wrapping:

Listing 1: Parallelized code for the image warp.

```
int index, xp, yp, tx = width / 2, ty = height / 2;
float x, y, radius, theta, PI = 3.141527f, DRAD = 180.0f / PI;
#pragma omp parallel for \
shared(inputImage, outputImage, width, height) \
private(x, y, index, radius, theta, xp, yp)
for (yp = 0; yp < height; yp++) {
    for (xp = 0; xp < width; xp++) {
        index = xp + yp * width;
        radius = sqrtf((xp - tx) * (xp - tx) + (yp - ty) * (yp - ty));
        theta = (radius / 2) * DRAD;
        x = cos(theta) * (xp - tx) - sin(theta) * (yp - ty) + tx;
        y = sin(theta) * (xp - tx) + cos(theta) * (yp - ty) + ty;
        outputImage[index] = BilinearlyInterpolate(inputImage, width, height, x, y);
    }
}
```


Paralelismo directo: Mandelbrot

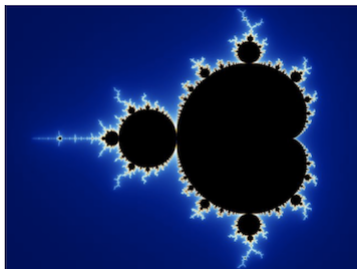
Es un set de

puntos complejos, que se calculan a través de la iteración de una función compleja no-divergente (iterada desde $z=0$) de forma

$$z_{k+1} = z_k^2 + c$$

donde z_{k+1} es la iteración (k+1) del número complejo $z = a + bi$ (con $i = \sqrt{-1}$), así como c es un número complejo. Se itera hasta

que la longitud del vector $z_{len} = \sqrt{z_{real}^2 + z_{imag}^2}$ es mayor que 2



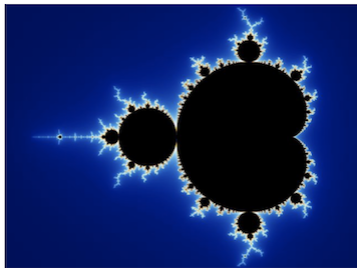
Paralelismo directo: Mandelbrot

En paralelo:

Cada pixel puede ser calculado independientemente del resto (el cálculo individual del pixel no es paralelizable)

Distribución estática de tareas:

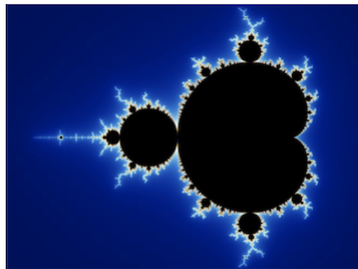
Cada pixel necesita un número distinto de iteraciones, por lo que cada proceso recibe una porción de la imagen, y no terminan sincronizadas entre ellas, lo que torna ineficiente al algoritmo.



Paralelismo directo: Mandelbrot

Distribución dinámica de tareas: Las tareas en paralelo se distribuyen en un área común (work pool), desde donde cada proceso obtiene una nueva tarea en caso quede inactivo

Dados n pixel y un número máximo de iteraciones que depende de c , $t \leq \text{max_iter} \times n$, y por ello el tiempo es $O(n)$



Paralelismo directo: Mandelbrot

Existen 3 pasos:

- Transferencia

de datos a $p-1$ procesos,

$$t_{comm_1} = (p - 1)(t_{startup} + t_{data})$$

- Cálculo

en paralelo $\cdot t_{comp} \leq \frac{\max_iter \times n}{p-1}$

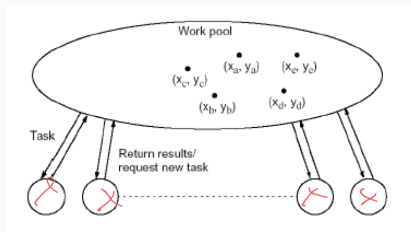
- Transferencia de resultados

al master constante, $t_{comm_2} = k$

En total

$$t_p = t_{comp} + t_{comm_1} + t_{comm_2}$$

El ratio cálculo/comunicación (granularidad) es $O(n/p^2)$



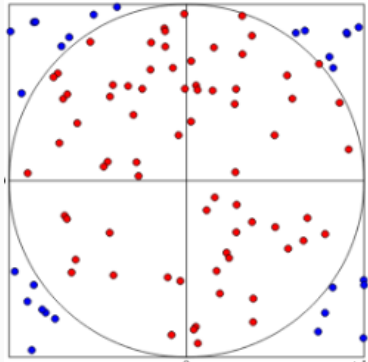
Uso de random en paralelo

Utiliza principios de cálculo probabilístico y estadística, para aproximar soluciones de problemas complejos. Por ello, los resultados son correctos con cierta probabilidad. Se utilizan:

- Física e ingeniería
- Computación gráfica y diseño de productos
- Finanzas
- Estudio de sismos o fenómenos naturales, clima

Método Montecarlo para calculo de PI

- Genera N puntos random en el cuadrado exterior
- Determina el número de puntos n dentro del círculo
- Siendo el área del círculo $\pi * r^2$, y el área del cuadrado $4 * r^2$
- Entonces $n/N = \pi/4$, o $\pi = 4 * n/N$
- Generando un número suficiente de puntos se logra un resultado mas exacto
- $10^3 < N < 10^6$



Método Montecarlo para calculo de PI (secuencial)

```
for(i=0; i<num_trials ; i++)
{
    x  = (double)rand() / RAND_MAX;
    y  = (double)rand() / RAND_MAX;
    test = x*x + y*y;
    if (test <= r*r) Ncirc++;
}

pi = 4.0 * ((double)Ncirc/((double)num_trials);
```

Método Montecarlo para calculo de PI (paralelo-OMP)

```
//Init Parallelization with reduction
#pragma omp parallel for reduction(+: count)
for(i=0; i<num_trials ; i++)
{
    x  = (double)rand() / RAND_MAX;
    y  = (double)rand() / RAND_MAX;
    test = x*x + y*y;
    if (test <= r*r) Ncirc++;
}

pi = 4.0 * ((double)Ncirc/((double)num_trials));
```

Método Montecarlo para calculo de PI

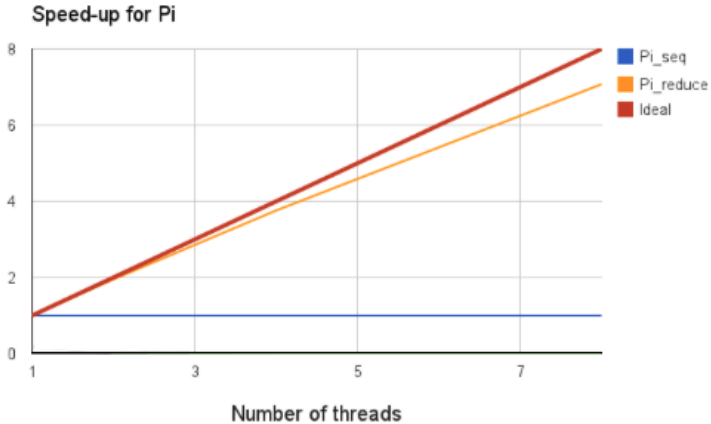
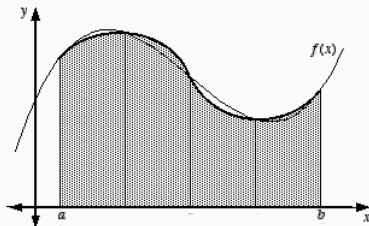


Figura 1: Speedup para el calculo de Pi

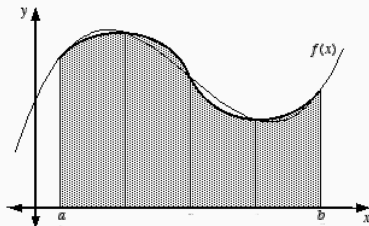
Método Montecarlo: Integración numérica

Dada una integral $I = \int_a^b f(x)dx$, ésta puede calcularse a través de la generación de N valores aleatorios entre a y b , y ser aproximados con $I_{MC} = \sum_N f(x)$. Ya que las iteraciones para suma o integrales son independientes, se paralelizan fácilmente, asignando a cada proceso un conjunto de números random entre a y b .



Método Montecarlo: Integración numérica

Estas se ejecutarán en cada proceso hasta que el criterio de aproximación se cumpla. Luego se recopila y suma la información en el nodo maestro. La generación aleatoria puede delegarse a cada proceso, evitando así tiempos de comunicación.

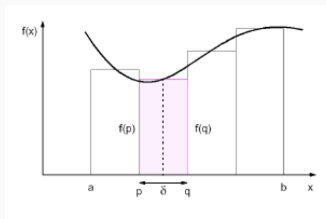


Integración numérica

Para el cálculo de una integral de la forma $I = \int_a^b f(x)dx$ se puede dividir el intervalo a-b en partes, tal que cada una se aproxima con un rectángulo, asignado a cada proceso.

Método estático:

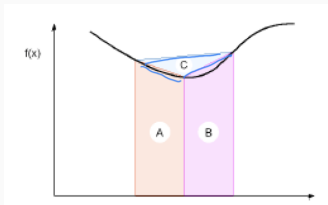
En el método del trapecio, cada área se calcula con la fórmula $\frac{1}{2}(f(p) + f(q))\delta$. Si se conoce δ de antemano, se comunica a cada proceso. Con p procesos, es $(b-a)/p$



```
wtime = omp_get_wtime();
total = 0.0;
# pragma omp parallel shared(a,b,n) private (i,x)
# pragma omp for reduction (+ : total)
for (int i = 0; i < n; i++) {
    x = ((double)(n-i-1)*a + (double)(i)*b) / (double)(n-1);
    total = total + f(x);
}
mtime = omp_get_wtime ( ) - wtime;
total = (b-a) * total / (double)n;
error = fabs (total - exact);
```

Cuadratura adaptativa:

Si no se conoce δ de antemano, se calcula hasta que se logre la convergencia deseada. δ se divide en 3 zonas, y se divide hasta que C sea lo suficientemente pequeño



Generación random en paralelo

El generador 'linear-congruente' produce números pseudo-aleatorios, utilizando la función $x_{i+1} = (ax_i + c) \bmod(m)$

Con a,c y m constantes, e.g. $a=16807$, $m=2^{31}-1$, $c=0$

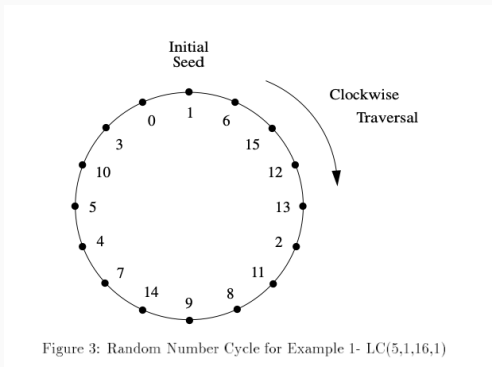


Figura 2: Ejemplo para $a=5$, $m=2^4$, $c=1$, $x_0 = 1$

- Estrategia centralizada: el maestro distribuye random a los procesos (bajo performance)
- Estrategia replicativa: cada proceso genera su propio seed, usando el mismo generador random (puede crear correlaciones)
- Estrategia distribuída: cada proceso genera su propio seed, usando su propio generador random (eficiente pero difícil de implementar)

Dados p procesos, se necesitan p independientes data streams de números random. Para ello se particiona el ciclo en segmentos independientes, cada uno accesible por un proceso distinto. Para $c=0$, se obtiene:

$$x_{n+1} = ax_n(mod m)$$

$$x_{n+2} = ax_{n+1} = a^2x_n(mod m)$$

$$x_{n+3} = ax_{n+2} = a^3x_n(mod m)$$

...

$$x_{n+k} = ax_{n+k-1} = a^kx_n(mod m)$$

Random tree en paralelo

Utiliza dos generadores (LGC), L y R

$$L(x) = x_L = a_L x + c_L(\text{mod } m)$$

$$R(x) = x_R = a_R x + c_R(\text{mod } m)$$

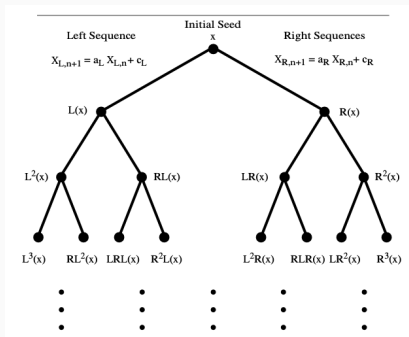


Figura 3: Operaciones del Random tree con respecto a x_0

RNG en paralelo

Implementa el generador recursivo múltiple MRG32k3a propuesto por L'Ecuyer (1999). Este provee de $1.8 \cdot 10^{19}$ random streams independientes, cada uno conteniendo $2.3 \cdot 10^{15}$ substreams. Cada substream tiene un período de $7.6 \cdot 10^{22}$. El periodo del generador en conjunto es de $3.1 \cdot 10^{57}$

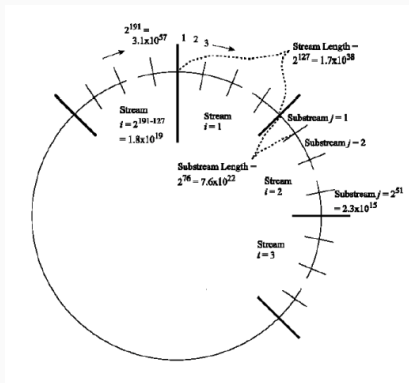


Figura 4: Streams y substreams

RNG en paralelo (C++)

Se implementa la clase RNG, con metodos como:

- `void set_seed (long seed)`, seed del RNG
- `long seed (void)`, retorna el seed actual
- `long next (void)`, retorna el siguiente random como un entero en $[0, MAXINT]$
- `double next_double (void)`, retorna el siguiente random en $[0, 1]$
- `void reset_start_substream (void)`
- `void reset_next_substream (void)`
- `double uniform (double r)`, retorna un numero de una distribución uniforme en $[0, r]$
- `double exponential (double k)`, retorna un numero de una distribución exponencial
- `double normal (double avg, double std)`, retorna un numero de una distribución normal

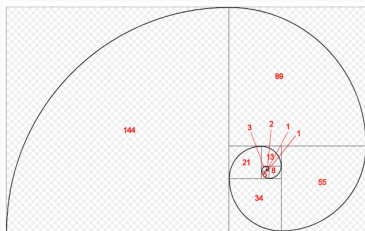
RNG en paralelo (C++): ejemplo

```
/* create new RNGs */
RNG arrival (23456);
RNG size;
/* set the RNGs to the appropriate substream */
for (int i = 1; i < 3; i++) {
    arrival.reset_next_substream();
    size.reset_next_substream();
}
/* print the first 5 arrival times and sizes */
for (int j = 0; j < 5; j++) {
    printf ("% -8.3f  % -4d\n", arrival.lognormal(5, 0.1),
            int(size.normal(500, 10)));
}
```

Secuencia Fibonacci

Se utiliza un algoritmo recursivo






```
funcion fib(n){  
  if(n<2)  
    return n;  
  return fib(n-1)+fib(n-2);  
}
```



Dynamic multithreading:

Aquí, **spawn** paraleliza $\text{Fibonacci}(n-1)$, **sync** sincroniza para garantizar se usen los valores correctos de x, y

```
Fibonacci(n):  
  if  $n < 2$ : | thread A  
    . return  $n$  | thread A  
   $x = \text{spawn Fibonacci}(n-1)$  | thread A  
   $y = \text{spawn Fibonacci}(n-2)$  | thread B  
  sync | thread C  
  return  $x+y$  | thread C
```


-  David B. Kirk and Wen-mei W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. 2nd. Morgan Kaufmann, 2013. isbn: 978-0-12-415992-1.
-  Norm Matloff. *Programming on Parallel Machines*. University of California, Davis, 2014.
-  Peter S. Pacheco. *An Introduction to Parallel Programming*. 1st. Morgan Kaufmann, 2011. isbn: 978-0-12-374260- 5.
-  Michael J. Quinn. *Parallel Programming in C with MPI and OpenMP*. 1st. McGraw-Hill Education Group, 2003. isbn: 0071232656.
-  Jason Sanders and Edward Kandrot. *CUDA by Example: An Introduction to General-Purpose GPU Program- ming*. 1st. Addison-Wesley Professional, 2010. isbn: 0131387685, 9780131387683.