



Computación Paralela y Distribuida

2022-II

José Fiestas

07/09/22

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

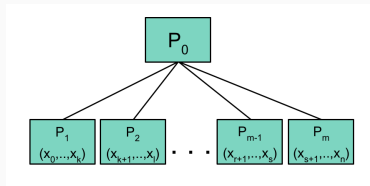
Particionamiento, Divide y Vencerás

Particionamiento

El dominio
se divide y los subdominios
se ejecutan independientemente.

Esto se aplica:

- A la data (data partitioning, domain decomposition)
- A las funciones del programa (functional decomposition). Aquí la comunicación es, generalmente, lo mas costoso



Ejemplo: suma de elementos de un array

El cálculo secuencial necesita $n-1$ adiciones, con complejidad $O(n)$.

Con p procesos se obtiene:

- Envío de data $t_{comm_1} = p(t_{startup} + (n/p)t_{data})$
- Cálculo de cada proceso $t_{comp_1} = n/(p - 1)$
- Recopilación de sumas parciales $t_{comm_2} = t_{startup} + pt_{data}$
- Suma total $t_{comp_2} = p - 1$

Complejidad es $t_p = p(t_{startup}) + (n + p)t_{data} + n/p$

Cálculo de fuerzas entre cuerpos, aplicado a problemas estocásticos (astrofísica, dinámica molecular).

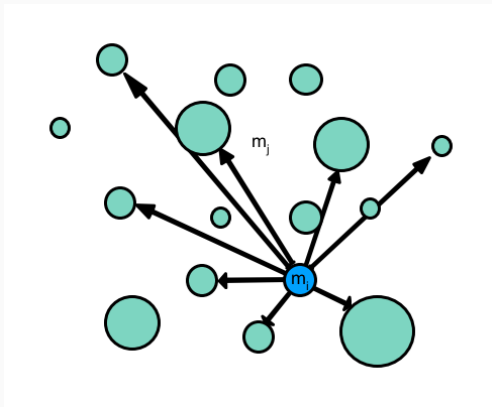
E.g. Gravitación en N-cuerpos

Se trata de calcular las fuerzas gravitatorias entre cuerpos en el espacio (planetas, estrellas, galaxias), con lo que se obtendrán posiciones y velocidades de los cuerpos en un tiempo determinado.

Problema de N-cuerpos

No hay solución analítica desde $N=3$. El cálculo es de la fuerza ejercida sobre el cuerpo i es

$$\mathbf{F}_i = -G\mathbf{m}_i \sum_{1 \leq j \leq N, j \neq i} \frac{\mathbf{m}_j \mathbf{r}_{ij}}{|\mathbf{r}_{ij}|^3}$$



Número de operaciones de cálculo de fuerza es $N(N-1)/2$

Problema de N-cuerpos

Para una simulación se puede solo aproximar el movimiento en intervalos pequeños. Sea el intervalo Δt , es para $t+1$

$F = \frac{m(v_{t+1}-v_t)}{\Delta t}$, y con ello, para la velocidad y la posición:

$$v_{t+1} = v_t + \frac{F\Delta t}{m}$$

$$x_{t+1} = x_t + v\Delta t$$

Ya que el cuerpo se mueve a una nueva posición, se modifican su aceleración y fuerza, que deberá volver a calcularse.

Nota: Ya que la velocidad no es constante en Δt es solo una aproximación. Por ello se utilizan métodos de interpolación en el tiempo como 'leap-frog'

Problema de N-cuerpos: código secuencial

```
class Nbody{
public:
float pos[3][N];
float vel[3][N];
float m[N];
}

int main(int arg, char**argv){
...
// define class galaxy
Nbody galaxy;
...
// initialize properties
galaxy.init();
// integrate forces

galaxy.integr();
return 0;
}

void integr() {
...
// medir CPU time
start=clock();
force(n,pos,vel,m,dt);
// medir CPU time
end=clock();
cpuTime=difftime(end,start)/
(CLOCKS\_PER\_SEC)
...
}
```

Problema de N-cuerpos: código secuencial

```
void force(int n, float pos[][],float vel[][], float m[], float dt) {
    ...
    // suma en i
    for (int i=0;i<n;i++)
    {
        float my_r_x=pos[0][i];
        // suma en j
        for (int j=0;j<n;j++)
        {
            if(j!=i) // evitar i=j
            {
                //calcular aceleracion
                float d=pos[0][j]-my_r_x; // 1 FLOP
                a_x+= G*m[j]/(d*d); // 4 FLOPS
                ...
            }
        }
        // actualizar velocidades
        vel[0][i] += a_x*dt; // 2 FLOPS
        // actualizar posiciones
        pos[0][i]+=vel[0][i]*dt; // 2 FLOPS
    }
}
```

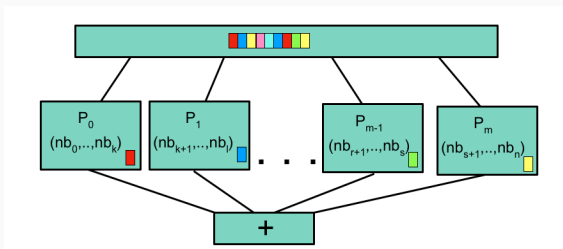
Problema de N-cuerpos en paralelo

Se puede realizar con partición directa, donde cada proceso se encarga de un grupo de cuerpos y las fuerzas son comunicadas con los otros procesos. La complejidad es $O(n^2)$, ya que cada cuerpo es afectado por los $n-1$ restantes. Esto no es eficiente.

En la práctica, se considera que un cluster de cuerpos distantes afecta aproximadamente como un solo cuerpo. El algoritmo de Barnes-Hut es de divide-and-conquer, que funciona dividiendo el dominio en cubos (en 3D) y subdividiéndolos en grupos de 8.

Problema de N-cuerpos en paralelo

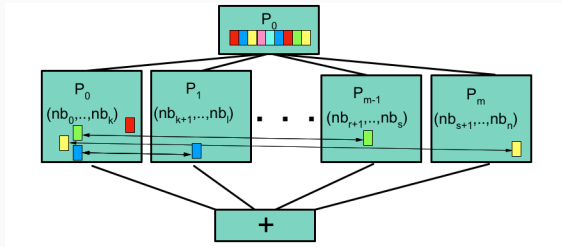
El problema de N-cuerpos se puede paralelizar utilizando **memoria compartida** (OpenMP/CUDA).



Usando **memoria compartida** se evita el tiempo de comunicación entre procesos, pero el límite es dado por la cantidad de núcleos de un procesador.

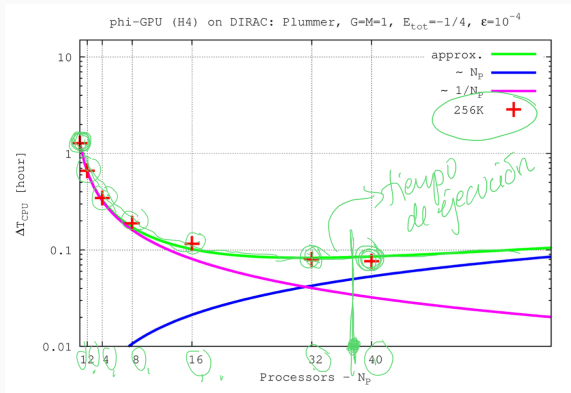
Problema de N-cuerpos en paralelo

El problema de N-cuerpos se puede paralelizar utilizando **memoria distribuída** (MPI).



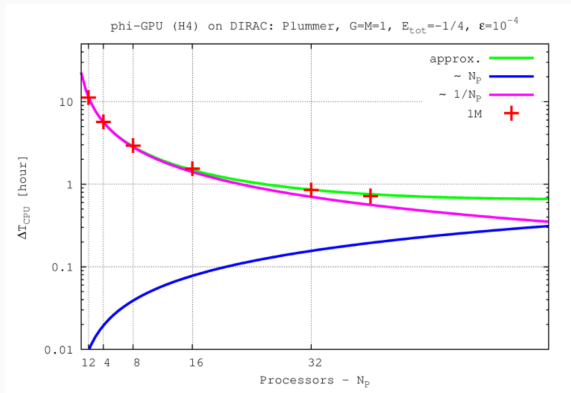
Usando **memoria distribuída**, se paga el precio de la comunicación entre procesos pero se puede siempre escalar el cluster para obtener mayor eficiencia.

Problema de N-cuerpos en paralelo



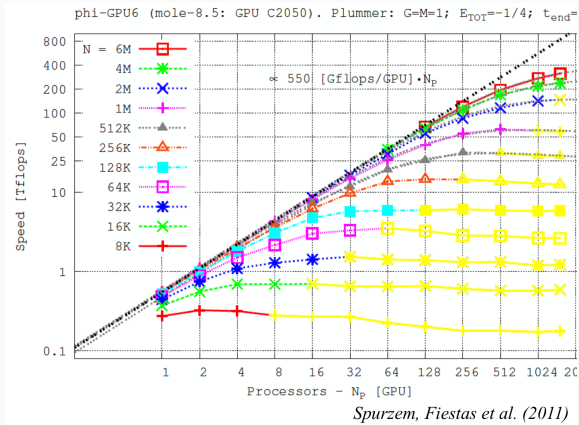
Tiempo de ejecución (en horas) vs. procesos (N_p). Los datos experimentales (cruces rojas) reproducen bien la curva teórica. T_{total} en azul, T_{computo} en magenta, T_{comm} en azul. $N=256K$

Problema de N-cuerpos en paralelo



Tiempo de ejecución (en horas) vs. procesos (N_P). Los datos experimentales (cruces rojas) reproducen bien la curva teórica. T_{total} en azul, T_{computo} en magenta, T_{comm} en azul. $N=10^6$

Problema de N-cuerpos en paralelo



Speedup vs. procesos para el problema de N-cuerpos. Para distintos valores de N .

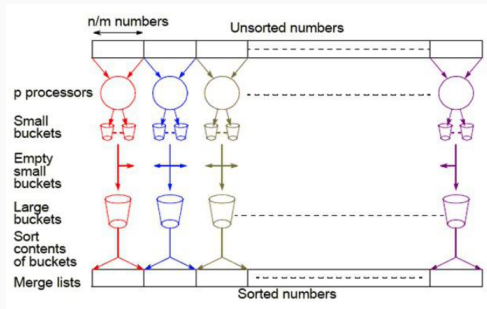
Particionamiento: Bucket Sort

Utiliza un método de **particionamiento**. Funciona mejor con conjuntos homogéneos. Un intervalo a se separa en m regiones (buckets), con n/m valores en cada una. Luego son los grupos en cada bucket ordenados (con quicksort o mergesort), en **$O((n/m) \log(n/m))$** y los resultados concatenados en una lista ordenada.

Tiempo secuencial: $t_s = n + m(\frac{n}{m} \log(\frac{n}{m})) = O(n \log(\frac{n}{m}))$

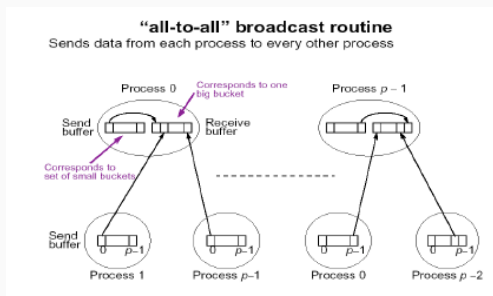
Ejemplo: Bucket Sort en paralelo

Cada bucket pertenece a un proceso, con lo que se logra $O((n/p) \log(n/p))$, con $p=m$ procesos. Pero cada proceso debe controlar todos los números no-ordenados para su selección. Si se separa la lista en p regiones, cada una para cada proceso, esta se separa en cada bucket y se envía luego a cada otro proceso, para ser ordenada finalmente ahí. Esto implica una comunicación all-to-all



Ejemplo: Bucket Sort en paralelo

Cada bucket pertenece a un proceso, con lo que se logra $O((n/p) \log(n/p))$, con $p=m$ procesos. Pero cada proceso debe controlar todos los números no-ordenados para su selección. Si se separa la lista en p regiones, cada una para cada proceso, esta se separa en cada bucket y se envía luego a cada otro proceso, para ser ordenada finalmente ahí. Esto implica una comunicación all-to-all



Ejemplo: Bucket Sort en paralelo

Primero se envía la data a los procesos y estos la clasifican. De ahí, los pasos son:

- 1. comunicación, $t_{comm_1} = t_{startup} + nt_{data}$
- 1. cálculo, n/p números en p buckets pequeños (en cada proceso),
 $t_{comp_1} = \frac{n}{p}$
- 2. comunicación, $p-1$ buckets son enviados a $p-1$ procesos,
 $t_{comm_2} = p(p-1)(t_{startup} + \frac{n}{p}t_{data})$
- 2. cálculo, los n/p números en los buckets grandes se ordenan,
 $t_{comp_2} = \frac{n}{p} \log(\frac{n}{p})$

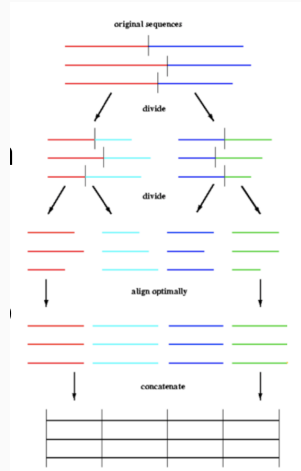
$$t_p = \frac{n}{p} + \frac{n}{p} \log(\frac{n}{p}) + nt_{data} + p(p-1)(t_{startup} + \frac{n}{p}t_{data})$$

Paradigma Divide y vencerás (divide and conquer)

Algoritmo heurístico

acelerado para solución de secuencias múltiples y homólogas. Funciona:

- Separando las secuencias en partes reduciendo su longitud. Esto es, dividiendo el problema en sub-problemas
- Optimizar su distribución o resolver los sub-problemas (recursiva, directamente)
- Concatenar resultados o combinar soluciones de sub-problemas en solución general



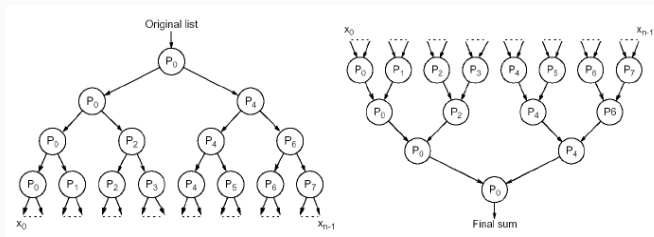
Paradigma Divide y vencerás (divide and conquer)

Se utiliza para operaciones globales en listas (ordenamiento), cálculo de máximo/mínimo, o búsqueda

En su versión recursiva, se generan dos sub-problemas en cada separación. Esto se representa con un árbol, que se divide para abajo y se recopila para arriba.

Paradigma Divide y vencerás (divide and conquer)

Paralelismo: Partes distintas del árbol pueden ser ejecutadas a la vez. Una solución sería asignar un proceso a cada nodo del árbol. Pero sería ineficiente, ya que para 2^m sub-problemas se necesitan $2^{m+1} - 1$ procesos



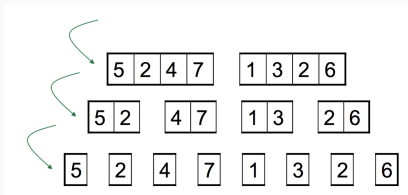
Ejemplo: Ordenamiento

Dado un array, **objetivo** es ordenar el array

5	2	4	7	1	3	2	6
---	---	---	---	---	---	---	---

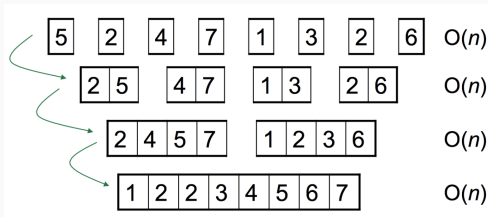
1	2	2	3	4	5	6	7
---	---	---	---	---	---	---	---

Paso 1: dividir el array. Necesita $\log(n)$ divisiones para lograr arrays de un elemento



Ejemplo: Ordenamiento

Paso 2: Conquista, requiere $\log(n)$ iteraciones, cada una tomando tiempo $O(n)$. $T(n) = O(n \log n)$

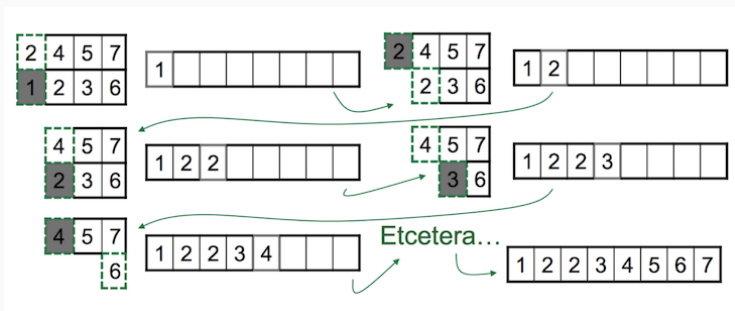


Paso 3: Combinar. Para 2 arrays de tamaño 1 la union es directa. En general, 2 arrays ordenadas de tamaño n y m pueden ser combinadas en un tiempo $O(n+m)$ para formar un array $n+m$ ordenada.



Ejemplo: Ordenamiento

Caso: Combinar dos arrays de tamaño 4



Ordenamiento: Mergesort

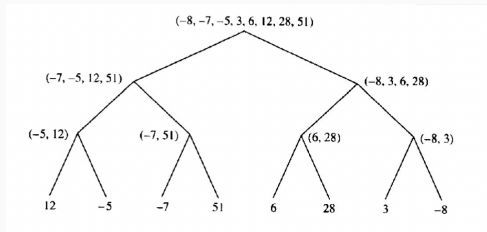
Ingreso: vector X con $n = 2^k$ números

Salida : árbol binario con n hojas, tal que para cada $0 < h < \log(n)$, $L(h,j)$ contiene la subsecuencia ordenada de los elementos contenidos en el árbol con el nodo (h,j) como raíz $1 < j < n/2^h$






Pseudocódigo:

1. **for** $i = 1$ to n **pardo**
 $L[0,i] := X[i]$
endfor
2. **for** $h = 1$ to $\log(n)$ **do**
 for $j = 1$ to $n/2^h$ **pardo**
 merge $L(h-1, 2j-1)$ y $L(h-1, 2j)$ en $L(h,j)$
 endfor
end

Ordenamiento: Mergesort



El paso 1 es $O(1)$, el paso 2 repite el **merge** un total de $\log(n)$ veces. Considerando un algoritmo óptimo en paralelo que mezcle dos arrays ordenados en otro con una complejidad $O(\log(\log(n)))$ (lo veremos en la unidad 5, Algoritmos), tenemos $T_p(n) = O(\log(n) \log(\log(n)))$. Siendo $T_s(n) = O(n \log(n))$.

-  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 Programming*. 1st. Addison-Wesley Professional, 2010. isbn: 0131387685, 9780131387683.