



Programación Paralela

2020-II

José Fiestas

November 28, 2020

Universidad Nacional de Ingeniería
jose.fiestas@uni.edu.pe

Unidad 2: Metodos de paralelismo

Objetivos:

1. Velocidad, eficiencia, escalabilidad. Ley de Ahmdal
2. DAG (Directed Acyclic Graphs)
3. Caso práctico: Algoritmo de N-cuerpos en paralelo
4. Operaciones basicas de paralelismo
5. Modelos computacionales en paralelo (PRAM)
6. Broadcast/Reduccion

Operaciones basicas de paralelismo

Secuencias: $a_n : \{a_0, a_1, \dots, a_{n-1}\}$

- **longitud(a)** $a = |a|$, $w=s=O(1)$
- **subsecuencia(a,i,j)**: $a = a[i, \dots, j]$, $w=s=O(1)$
- **splitmid(a)** : $(sp[0, \frac{n}{2}-1], sp[\frac{n}{2}, n-1])$, $w=O(n)$, $s=O(1)$

Tabulamiento (de una secuencia): **tab**(f(x),int,seq)

e.g. **tab**(i,n,seq), $w=O(n)$, $s=O(1)$

En general: $w = \sum_i w(f(i))$, $s = MAX(S(f(i)))$

- secuencia vacía: **tab**(f,0)
- secuencia identidad e: **tab**(f,1)
- mapping(f,a): **tab**(f(a[i]), |a|)
- append(a,b): **tab** (if $i < |a|$ then a[i] else b[i - |a|]), $w= O(|a| + |b|)$, $s=O(1)$

Iteración (de una secuencia): $\text{iter}(b, f(a_i) \rightarrow x + a_i, |a|)$

E.g. $\text{iter}(0, f(a_i) \rightarrow s + a_i, |a|) \rightarrow (((((0+1)+2)+3)\dots+(n-1)) \rightarrow (((a_0 + a_1) + a_2) + a_3)\dots + a_{n-1})$ (suma acumulada)

E.g. la función que mapee el valor de una lista al inmediato anterior, que no sea cero.

$(1,0,4,5,2,0,0,3,4) \rightarrow (0,1,1,4,5,2,2,2,3)$

$\text{fun skipcero}(x,y) := \text{if } x > 0 \text{ then } x, \text{ else } y$

isort (a): **iter**() insert a

fun isort(x,r) = iter()

$w = \sum_i (W(f(x_i), a[i])), s = \sum_i (S(f(x_i), a[i])) = O(n^2)$

```
reduce (f,a): if  $|a| = 0$  then id  
else if  $|a| = 1$  then  $a[0]$ ,  
else (b,c)=split_mid(a)  
(rb,rc)= reduce b || reduce c  
return (rb,rc)
```

$$W(n) = 2w\left(\frac{n}{2}\right) + O(1) = O(n)$$

$$S(n) = \text{MAX}(S\left(\frac{n}{2}\right) + O(1)) = \log(n)$$

dada una función de complejidad constante $O(f(n)) = O(1)$

PRAM

Clasificación segun acceso en paralelo a registros de memoria:

{exclusive, concurrent} **read** {exclusive, concurrent} **write**

- **EREW**, cada locación de memoria puede ser leída o escrita por un solo proceso
- **CREW**, múltiples procesos pueden leer la memoria pero solo uno puede escribir en ella.
- **ERCW**, no es usada
- **CRCW**, múltiples procesos pueden leer y escribir

Modelos Formales:

E.g.

Sea una instrucción de la forma:

Procesador i : $c := a + b$

donde a , b , y c son espacios de memoria compartidos.

Las instrucciones para el procesador i serán:

- Copiar espacio a en la memoria local
- Copiar espacio b en la memoria local
- Sumar a y b
- Escribir el resultado en el espacio c

Indicadores en PRAM:

Tiempo de ejecución (paralelo), $T(n)$: designa el número máximo de ordenes que ejecuta un proceso (número de iteraciones)

Número de procesos activos, $P(n)$

Espacio de memoria $M(n)$: registros de memoria utilizados.

Trabajo, $W(n)$: suma de ordenes por proceso.

Costos máximos $W(n) \sim T(n) P(n)$

El máximo valor es **Worst Case Complexity**

Suma de n números:

Si cada proceso suma dos elementos, el número de elementos se reduce a la mitad en cada iteración, por lo tanto se necesitan **$\log(n)$** pasos para la suma de n números.

Además, se requieren **$O(n)$** procesos.

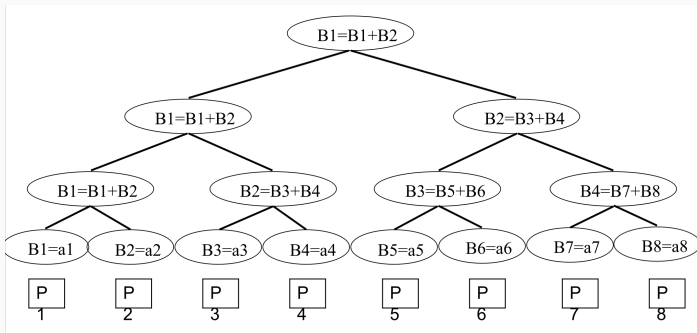
I.e.

$$T(n) = O(\log n)$$

$$P(n) = O(n)$$

$$W(n) = O(n \log n)$$

Para $n=8$, $np=8$



Multiplicación matriz-vector

Ingreso: $A(n,n)$, $x(n)$, p procesos i , $r=n/p$

Salida: componentes $[ir, \dots, (i+1)r-1]$ del vector y

Pseudocódigo para el proceso i :

1. $z := X$; **Globalread**(X, z)
2. $b := A[ir : (i+1)r-1, 1 : n]$; **Globalread**(A, b)
3. $w := b \cdot z$; **Cálculo**
4. $Y[ir : (i+1)r-1] := w$; **Globalwrite**(w, Y)

Máximo de n números

Se distribuyen n procesos a cada elemento de un array A con n números.

1. Cada proceso compara el valor del elemento con el resto, y determina si ese elemento es el mayor del array A (puede usar un array booleana con 0 o 1)
2. El índice del elemento de A donde el array booleano sea 1 será el máximo de A

Es decir, el máximo de n números se puede calcular en $O(1)$ usando n^2 procesadores, lo que lo hace impracticable para n grande.

En forma **secuencial**, $P(n)=1$, $T(n) = O(n) \rightarrow W(n)=O(n)$,

En **paralelo**, $P(n)=O(n^2)$, $T(n) = O(1) \rightarrow W(n)=O(n^2)$.

Se optimiza solo $T(n)$ sin considerar $P(n)$, o $W(n)$.

Por consiguiente, un algoritmo es **óptimo** cuando **$W(n)$** es óptimo.

Normalmente comparado con el algoritmo secuencial

Hay problemas fáciles de paralelizar, y problemas imposibles de paralelizar. Un objetivo es la aceleración de la ejecución

Si $T(n,1)$ es el costo del algoritmo con un procesador, y $T(n,p)$ el costo de p procesadores, la velocidad está definida como: $S(p) = \frac{T(n,1)}{T(n,p)}$

$S(p)$ es óptima, si $S(p) = p$

La eficiencia da la carga promedio de los procesadores

$$E(n, p) = \frac{S(p)}{p} = \frac{T(n,1)}{pT(n,p)}$$

Lenguaje formal utiliza **pardo** (do in parallel)

```
for  $p_i, 1 \leq i \leq n$  pardo  
  A(i) := B(i)
```

Es decir, n operaciones serán ejecutadas en paralelo, i.e. proceso P1 asigna B(1) a A(1), proceso P2 asigna B(2) a A(2), etc

Suma de n números

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

Salida : suma S de los números del vector

Pseudocódigo:

1. **for** $i = 1$ to n **pardo**
2. $B[i] = A[i]$
3. **endfor**
4. **for** $h = 1$ to $\log(n)$ **do**
5. **for** $i = 1$ to $n/2^h$ **pardo**
6. $B[i] = B[2i-1] + B[2i]$
7. **endfor**
8. **end**
9. $S := B(1)$

$$T(n) = O(\log n)$$

$$P(n) = O(n)$$

$$W(n) = O(n \log n)$$

Un modelo computacional en paralelo es una abstracción de la funcionalidad de un sistema de acceso, manejo y almacenamiento de información, pero que es independiente del hardware/software específico que se utiliza.

Modelo PRAM:

El modelo **RAM** (Random Access Machine, modelo Von Neumann) es una abstracción de un computador secuencial con memoria infinita, en el cual cada espacio de memoria puede ser accedida en forma directa. Contiene instrucciones para lectura, escritura, y operaciones aritméticas/lógicas. Es útil para el análisis de algoritmos secuenciales.

El modelo **PRAM** (Parallel Random Access Machine), consiste en un conjunto de procesadores idénticos $\{P_1, P_2, \dots, P_n\}$, con acceso a una memoria compartida para lectura/escritura de datos. Cada procesador es un RAM, que ejecuta el mismo código en forma sincrónica.

Cada proceso, en cada paso , ejecuta:

- lectura de data de la memoria común
- ejecución local de instrucciones
- escritura de data en la memoria común

No existe una red entre procesos, si no que se comunican a través de la memoria común.

Clasificación segun acceso en paralelo a registros de memoria:

{exclusive, concurrent} **read** {exclusive, concurrent} **write**

- **EREW**, cada locación de memoria puede ser leída o escrita por un solo proceso a la vez
- **CREW**, múltiples procesos pueden leer la memoria pero solo uno puede escribir en ella durante el mismo paso (Broadcast).
- **ERCW**, se permite escritura simultánea, pero la lectura es exclusiva en el mismo paso. No es usada
- **CRCW**, múltiples procesos pueden leer y escribir en el mismo paso

Escritura simultánea exige resolver el problema de escritura en el mismo espacio de memoria:

- **modelo común**: todos los procesos escriben la misma data
- **modelo arbitrario**: todos los procesos escriben un dato arbitrario en el mismo paso
- **modelo combinado**: ejecuta la acumulación de los resultados escritos arbitrariamente por cada proceso en el mismo espacio de memoria y en el mismo paso.
- **modelo prioritario**: los procesos tienen una prioridad de acceso, y aquel con la mayor prioridad podrá escribir en memoria.

El costo de un algoritmo PRAM se define como la cantidad de pasos necesarios para completar el programa. Cada paso puede ser de lectura, escritura o de computación local.

Global **OR**:

Entrada en $x[1, \dots, p]$

Result=0

for $i=1, \dots, p$ **pardo**
 if $x[i]$ **then** Result:=1

Global **AND**:

Entrada en $x[1, \dots, p]$

Result=1

for $i=1, \dots, p$ **pardo**
 if not $x[i]$ **then** Result:=0

Entrada en $A[1, \dots, n]$

Salida en $M[1, \dots, n]$ $M[i] = 1$, iff $A[i] = \max_j A[j]$

forall $(i, j) \in \{1, \dots, n\}^2$ **pardo**

$B[i, j] := A[i] \geq A[j]$

forall $i \in \{1, \dots, n\}$ **pardo**

$M[i] = \bigwedge_{j=1}^n B[i, j]$

Con $P(n) = n^2$

$T(n) = 1$

$S(n) = T(1)/T(n) = n$

$E(n) = 1/n$

Para emular más cercanamente a un computador real, se han creado varios modelos PRAM, como:

- **SB-PRAM**, que utiliza operadores lógicos para acceder a la memoria común simulados bajo un esquema round-robin (equitativo). Estas máquinas son un ejemplo de procesadores multithreading.
- **phase PRAM**, en el que las operaciones están divididas en fases, tal que los procesos pueden trabajar sin sincronización en las fases, pero son sincronizados al final de las fases
- **delay PRAM**, introduce un retraso entre el momento en que se produce la data en un proceso, y el momento en que es utilizada por otro proceso, con el objetivo de medir retrasos en acceso a memoria.
- **block PRAM**, donde el acceso. memoria toma un tiempo $l+b$, donde l es el tiempo de inicialización de la comunicación, y b es el tamaño del bloque de memoria accedido.

Modelo BSP (bulk synchronously parallel):

Es una computadora que consiste en procesos con memoria local y capaz de comunicarse punto a punto a través de una red.

La idea es que el software sea independiente de la arquitectura, y sea, por consecuencia, portable.

El cómputo está organizado en **supersteps**, que a su vez, constan de tres fases:

- cálculo simultáneo en cada proceso
- comunicación entre procesos
- barrera de sincronización que culmina la operación de comunicación y hace data visible a los procesos

Modelo BSP (bulk synchronously parallel):

Una unidad de cálculo se compone de:

- **p**: procesos virtuales, que ejecutan cálculos en cada superstep
- **s**: la velocidad de ejecución en número de operaciones (aritmética/lógica) por segundo, por proceso
- **l**: número de pasos necesarios por barrera de sincronización
- **g**: número de pasos promedio necesarios para transferir una palabra, tal que la ejecución de un proceso con m palabras, necesita $l \cdot m \cdot g$ pasos.

Modelo BSP (bulk synchronously parallel):

El tiempo

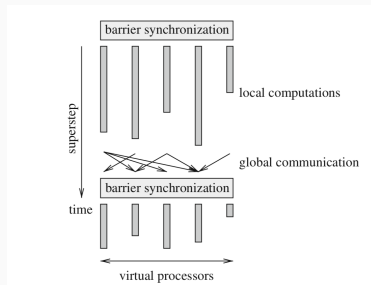
de ejecución de un superstep

($T_{superstep}$) esta dado por:

$$T_{superstep} = \max(w_i) + h \cdot g + l$$

Donde

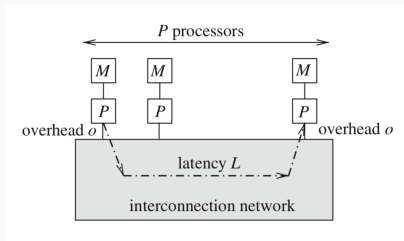
w_i es el tiempo necesitado por proceso i , $h \cdot g$ es el tiempo total de comunicación y l es el tiempo de la barrera de sincronización.



Modelo log P

Tiene la estructura de BSP, i.e. un sistema de memoria distribuido, con los siguientes parámetros:

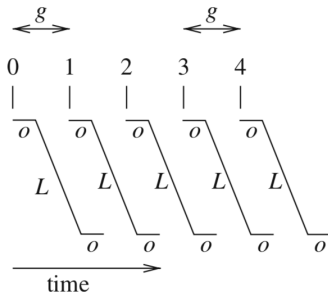
- **L** (latencia), es un límite superior de la latencia de la red, al enviar un mensaje pequeño.
- **o** (overhead), el tiempo requerido en comunicación, durante el cual no se puede ejecutar ninguna otra operación
- **g** (gap), es el tiempo mínimo entre envío/recibo de mensajes consecutivos por proceso
- **P** (procesos), el número de procesadores de la máquina



Modelo log P

El tiempo de ejecución de un algoritmo en el modelo **log P** es el máximo de los tiempos de ejecución entre todos los procesos

La transmisión de un mensaje largo se hace en varios mensajes pequeños.



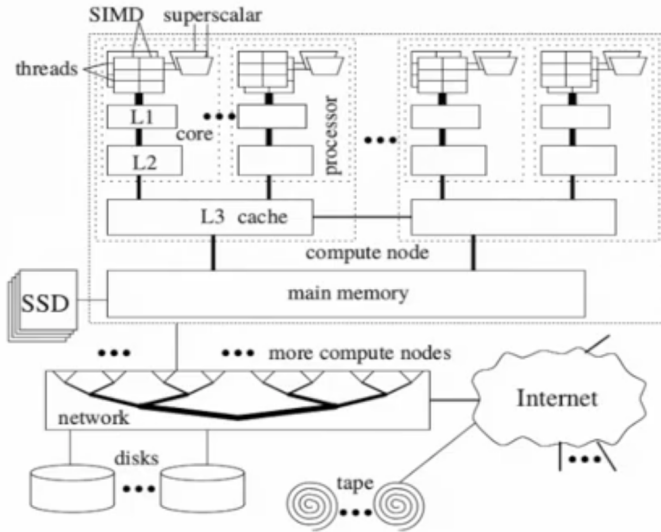
Modelo realístico:

- asincrónico
- CRQW (concurrent read, queued write), el costo de p procesos accediendo a un espacio de memoria simultáneamente, es $O(p)$
- Operaciones de escritura consistentes con instrucciones atómicas
- jerarquía en distribución de memoria

Estrategia es crear modelos con pocos niveles de paralelismo (híbridos)

Modelos Computacionales en Paralelo

Modelo Realístico.



DAG → **PRAM**:

Cada nivel del DAG que puede ser paralelizado, se representa como un modelo PRAM

DAG → Red:

- Nodos son procesos
- Aristas son líneas de comunicación en la red
- Tiempo de ejecución depende de los nodos, aristas, y de la longitud de las rutas en cada nivel del DAG.

PRAM → **DAG**:

Identificar bucles y condicionales con distintos procesos en PRAM

Sea \oplus un operador asociativo, que puede ser calculado en tiempo constante, se cumple que

$$\bigoplus_{i < n} x_i := (\dots((x_0 \oplus x_1) \oplus x_2) \oplus \dots \oplus x_{n-1})$$

Se calcula en tiempo $O(\log n)$ en PRAM, y en $O(T_{start} \log n)$ en un array lineal.

E.g., $+$, \cdot , \max , \min

Operaciones asociativas, reducción

código PRAM ($p=n$)

$p \in \{0, \dots, n-1\}$

activo:=1

for $0 \leq k < \log n$ **do**

if activo **then**

if bit k of i **then**

 activo=0

else if $i + 2^k < n$ **then**

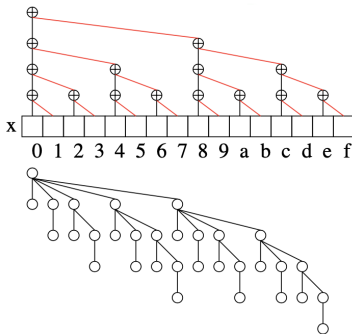
$x_i := x_i \oplus x_{i+2^k}$

los resultados aparecen en x_0

Dados n

procesos, $T=O(\log n)$, velocidad

$S=O(n/\log n)$, $E=O(1/\log n)$



Operaciones asociativas, reducción

$$p < n$$

n/p elementos a cada proceso

Entonces, la suma es

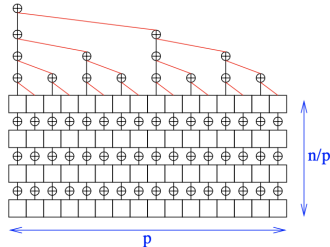
en paralelo de p sumas parciales

Tiempo = $T_{seq} (n/p) + O(\log p)$

Eficiencia:

$$\frac{T_{seq}}{p(T_{seq}(n/p) + O(\log p))} = \frac{1}{1 + O(p \log p)/n} = 1 - O\left(\frac{p \log p}{n}\right),$$

si $n \gg p \log p$



Operaciones asociativas, reducción

Memoria distribuída

$p \in \{0, \dots, n-1\}$

activo:=1

$s : x_i$

for $0 \leq k \leq \log n$ **do**

if activo **then**

if bit k of i **then**

sync-send s to $p \ i - 2^k$

 activo=0

else if $i + 2^k < n$ **then**

receive s' from $p \ i + 2^k$

$s := s \oplus s'$

los resultados aparecen en $p=0$

Comunicación total: $\Theta((T_{start} + T_{byte}) \log p)$

Array lineal: $\Theta(p)$, paso k necesita 2^k

BSP: $\Theta((l+g) \log p) = \Omega(\log^2 p)$

Ejercicios

Ejercicio 1

Genere un pseudocódigo secuencial y en paralelo (utilizando PRAM) para calcular la suma de elementos de un array con el método de Suma de Prefijos (suma acumulada)

Entrada: n números $a_1 \dots a_n$

Salida: Suma de $s_1 \dots s_n$, con $s_k = \sum_{i=1}^k a_i$

a: 2,5,2,3,4,5,2,1

s: 2,7,9,12,16,21,23,24

Determine $T(n)$, $P(n)$, $W(n)$ en forma secuencial y en paralelo.

Ejercicio 2

Desarrolle el algoritmo mergesort basado en las operaciones de iteración y reduce. Además determine $W(n)$ y $S(n)$

Diagrame el DAG correspondiente al siguiente código

```
double funcion()
{
    int i,n,j;
    double *v,*w,*z,sv,sw,x,res;

    /* Leer los vectores v, w, z, de dimension n */
    leer(&n, &v, &w, &z);





    calcula_v(n,v);           /* tarea 1 */
    calcula_w(n,w);           /* tarea 2 */
    calcula_z(n,z);           /* tarea 3 */

    /* tarea 4 */
    for (j=0; j<n; j++) {
        sv = 0;
        for (i=0; i<n; i++) sv = sv + v[i]*w[i];
        for (i=0; i<n; i++) v[i]=sv*v[i];
    }

    /* tarea 5 */
    for (j=0; j<n; j++) {
        sw = 0;
        for (i=0; i<n; i++) sw = sw + w[i]*z[i];
        for (i=0; i<n; i++) z[i]=sw*z[i];
    }

    /* tarea 6 */
    x = sv+sw;
    for (i=0; i<n; i++) res = res+x*z[i];

    return res;
}
```

-  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.