Ejercicios Algoritmos Paralelos

CC4102 - Diseño y Análisis de Algoritmos
Profesor: Gonzalo Navarro
Auxiliar: Jorge Bahamonde
Ayudantes : Sebastián Ferrada, Willy Maikowski

Ejercicios

P1. Defina claramente $T(n)$, $W(n)$, $T(n,p)$ y $S(n,p)$.

P2. ¿Qué significa eficiencia en el contexto de paralelismo? ¿Por qué no puede ser mayor que 1? Utilice esta propiedad para encontrar relaciones entre $T(n,p)$, $W(n)$ y $T(n)$.

P3. Se desea, dado un cierto valor x, generar n copias de éste (puede imaginar la inicialización de un arreglo como una aplicación de un algoritmo de este tipo).

1. Diseñe un algoritmo EREW de tiempo $O(\log n)$ que utilice n procesadores para este problema.
2. Mejórelo para obtener una eficiencia $\Theta(1)$.

P4. Considere paralelizar el problema de multiplicar dos matrices de $n \times n$. Considere que $T(n) = O(n^3)$: es decir, ignoramos algoritmos mejores que el estándar.

1. Proponga un algoritmo de tiempo $O(n)$ usando n^2 procesadores. Calcule $T(n,p)$, $W(n,p)$ y $E(n,p)$. ¿Qué modelo PRAM usa?
2. Proponga un algoritmo de tiempo $O(\log n)$ usando n^3 procesadores. Calcule $T(n,p)$, $W(n,p)$ y $E(n,p)$.
3. Mejore la eficiencia del algoritmo anterior para que sea $\Theta(1)$. ¿Qué tiempo obtiene y con cuántos procesadores?

P5. Considere un modelo PRAM CRCW en que cada vez que se produce una escritura concurrente de varios procesadores sobre una celda entonces todos estos procesadores deben escribir el mismo valor; en caso contrario, el comportamiento del modelo se indefine (es decir, le exigimos a los algoritmos que escriban de forma consistente).

Demuestre que cada paso realizado por un algoritmo CRCW con p procesadores bajo este modelo puede ser simulado con la misma cantidad de procesadores por un algoritmo CREW en tiempo $O(\log^2 p)$. *Hint:* utilice un arreglo auxiliar de tamaño p donde cada procesador escribe tanto el dato como la ubicación donde inicialmente se iba a escribir tal dato en el algoritmo CRCW.

P6. Sea $+$ una operación asociativa sobre un dominio D. $+$ no necesariamente representa la adición: utilizaremos este símbolo para aprovechar notación (por ejemplo, no podemos suponer que hay conmutatividad).

Considere arreglos $L = l_1 \dots l_n$ y $B = b_1 \dots b_n$ de elementos en D y $\{0,1\}$, respectivamente. Para cada $i \in [1,n]$ definimos $s(i)$ como el mayor $j \in [1,n]$ tal que $b_j = 1$.

Construya un algoritmo que compute los *prefijos determinados en L por B*. Esto corresponde al arreglo $P = p_1 \dots p_n$, donde $p_i = \sum_{s(i) \le j \le i} l_j$, con $\sum$ definida sobre la operación $+$. Su algoritmo debe trabajar en tiempo $O(\log n)$ y tener eficiencia $\Theta(1)$.

Soluciones

P1. Dado un problema:

- $T(n)$ es el tiempo de ejecución del *mejor* algoritmo secuencial conocido para el problema ante una instancia de éste de tamaño n.
- $W(n)$ es el trabajo total, definido como el número total de operaciones de los p procesadores involucrados en un algoritmo paralelo.
- $S(n,p)$ es el *speedup*, definido como $\frac{T(n)}{T(n,p)}$. Representa cuánto más rápido se resuelve el problema al utilizar paralelismo.

P2. La eficiencia se define como

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

¿Por qué no puede ser mayor que 1? Por contradicción, supongamos que existe un algoritmo paralelo A con $E_A(n,p) > 1$. Luego, utilizando la definición de speedup:

$$\begin{aligned} E_A(n,p) &= \frac{T(n)}{p \cdot T_A(n,p)} > 1 \\ \implies T(n) &> p \cdot T_A(n,p) \end{aligned}$$

Imaginemos una simulación de A que ejecuta cada paso de cada procesador de forma secuencial, utilizando un único procesador. Llamemos A' a este algoritmo. El tiempo que toma esta simulación es a lo más p veces lo que demora $T_A(n,p)$ (ya que algunos procesadores podrían terminar antes que otros):

$$T_{A'}(n) \leq p \cdot T_A(n,p) < T(n)$$

con lo que A' sería un algoritmo mejor que el óptimo, lo que es una contradicción. Por lo tanto, la eficiencia debe ser menor o igual a 1.

Finalmente, $W(n)$ es el trabajo hecho por todos los procesadores, con lo que:

$$p \cdot T(n,p) \geq W(n)$$

y, por un razonamiento análogo al de la demostración anterior,

$$E(n,p) = \frac{T(n,1)}{p \cdot T(n,p)} \leq \frac{T(n,1)}{W(n)} \leq 1$$

P3. 1. Se pide un algoritmo EREW: tanto las lecturas como las escrituras serán exclusivas. Si el algoritmo pudiera ser CREW, podríamos resolver el problema en $O(1)$: en un paso, los n procesadores copian el valor.

   El algoritmo que daremos no es complejo: consiste en duplicar la cantidad de copias del valor en cada paso.

Input: Un valor x; un arreglo B de tamaño n
Proc. 1**:**
| $B[0] \leftarrow x$
end
// s `representa cuántos valores están listos para ser copiados: se duplica en cada iteración`
$s \leftarrow 1$;
while $s \leq n$ **do**
| **Proc.** $j \in [0, \min(s, n-s)]$**:**
| | $B[j+s] \leftarrow B[j]$
| **end**
| $s \leftarrow 2 \cdot s$
end

Es claro que a lo más hay $\log n$ iteraciones, ya que s se duplica en cada paso y el algoritmo termina cuando $s > n$.

2. Una forma de mejorar nuestro algoritmo es usando el lema de Brent. Este dice que bajo condiciones generales, un algoritmo EREW de tiempo $T(n,p) = t$ donde el trabajo total ejecutado es W se puede convertir en uno que consigue:

$$T(n, \frac{W}{t}) = O(t)$$

En este caso, el trabajo total hecho por los procesadores es n (el valor x es copiado n veces) en tiempo $t = \log n$. Luego el lema de Brent dice que podemos construir un algoritmo tal que:

$$T(n, \frac{n}{\log n}) = \log n$$

La eficiencia de este algoritmo será:

$$\frac{T(n)}{p \cdot T(n,p)} = \frac{n}{\frac{n}{\log n} \cdot \log n} = 1$$

P4. Sabemos que el elemento C_{ij} de la matriz resultante de la multiplicación de dos matrices A y B puede calcularse como:

$$C_{ij} = \sum_k A_{ik} \cdot B_{kj}$$

Luego iterar sobre i, j y k nos da la solución estándar que toma $O(n^3)$ operaciones.

1. Tenemos n^2 procesadores a nuestra disposición, que denotaremos P_{ij}, para i y j en $\{1 \dots n\}$. Asignaremos cada uno de estos procesadores a una celda C_{ij} de C; en cada paso del algoritmo, le agregarán un término de la suma para C_{ij}:

Input: Dos matrices A y B, de dimensiones $n \times n$
Output: La matriz $C = A \cdot B$
Proc. P_{ij}**:**
| $C_{ij} \leftarrow 0$;
| **for** $k = 1 \dots n$ **do**
| | $C_{ij} \leftarrow C_{ij} + A_{ik} \cdot B_{kj}$
| **end**
end

Queda ver las propiedades del algoritmo:

- $T(n,p) = n$, pues cada procesador realiza n iteraciones.
- $W(n) = n^3$, pues cada procesador suma n productos. Como son n^2 procesadores, obtenemos n^3 operaciones en total.
- $E(n) = \frac{n^3}{n^2 \cdot n} = 1$.
- El modelo es CREW, pues dos procesadores pueden estar leyendo el mismo valor en A o en B: por ejemplo, P_{12} y P_{22} leen B_{k2} de manera concurrente, para $k = 1 \dots n$. La escritura se hace de forma exclusiva, pues cada procesador tiene asignado un lugar de escritura diferente.

2. Ahora tenemos n^3 procesadores, pero tenemos que reducir el tiempo a $O(\log n)$. Mirando de nuevo la definición que dimos para C_{ij}, una maniobra que se hace posible al tener n^3 procesadores es calcular todos los productos $A_{ik} \cdot B_{kj}$ en un paso, ya que son n^3. La suma posterior puede hacerse en tiempo $O(\log n)$: como la suma es una operación asociativa, podemos realizarla a través de árboles de sumas.
   Denotaremos los procesadores como P_{ikj}.

   Input: Dos matrices A y B, de dimensiones $n \times n$
   Output: La matriz $C = A \cdot B$
   `// Calculamos todos los productos`
   Proc. P_{ikj}**:**
   | $X_{ikj} \leftarrow A_{ik} \cdot B_{kj}$
   end
   `// Ahora realizamos` n^2 `sumas en ‘‘torneos’’: cada uno usa` n `procesadores. Esto toma` $O(\log n)$`: un paso por cada nivel del árbol de sumas.`
   Proc. $P_{i1j} \dots P_{inj}$**:**
   | $C_{ij} \leftarrow \textsc{Sum}_{P_{i1j} \dots P_{inj}}(X_{i1j} \dots X_{inj})$
   end

   Ahora tenemos que:

   - $T(n,p) = O(\log n)$, pues los productos se calculan en $O(1)$ y los árboles de sumas (o torneos) se calculan en tiempo $\log n$.
   - $W(n) = n^3$, idéntico al caso anterior (estamos haciendo las mismas operaciones).
   - $E(n) = \frac{n^3}{n^3 \cdot \log n} = \frac{1}{\log n}$.

3. Necesitamos aumentar la eficiencia de nuestro algoritmo. Si usáramos el Lema de Brent, llegaríamos a:

$$T(n, \frac{n^3}{\log n}) = O(\log n)$$

   con lo que obtenemos una eficiencia $\Theta(1)$. Sin embargo, el algoritmo que acabamos de formular está en el modelo CREW, pues al calcular los productos hay lecturas concurrentes. Para llevar el algoritmo al modelo EREW, haremos lo siguiente:

   - Antes de calcular los productos, crearemos n copias de A y B. Del problema **P1** sabemos que esto se puede hacer en tiempo $\log n$ si asignamos n procesadores para cada elemento de la matriz a copiar. Como queremos hacer copias de dos matrices, esto nos tomará $2 \log n$.
   - Con estas n copias, el cálculo de cada producto $A_{ik} \cdot Bkj$ puede realizarse leyendo de un par de matrices diferentes: lo que varía es el índice k. En otras palabras, dividimos los n^3 procesadores en grupos $1 \dots n$ de n^2 procesadores: el k-ésimo grupo calcula $A_{ik} \cdot B_{kj} \forall i, j$ utilizando el k-ésimo par de copias de A y B. Dentro de cada grupo no hay lecturas concurrentes, con lo que el algoritmo pasa al modelo EREW.

– Realizar las sumas a través de torneos es EREW :)

El algoritmo total sigue tomando tiempo $O(\log n)$, por lo que al utilizar el lema de Brent se llega a una eficiencia $\Theta(1)$.

P5. Para simular el algoritmo CRCW con uno CREW, sólo es necesario especificar cómo haremos las escrituras de modo de que el resultado sea el mismo que con el algoritmo CRCW original (el resto de las operaciones puede hacerse de forma idéntica en ambos modelos).

Consideremos un paso del algoritmo CRCW. Siguiendo el hint, llamaremos L como el arreglo que contiene los pares (l_i, v_i) de ubicaciones y valores que los p procesadores quieren escribir en este paso. Para uniformizar, podemos suponer que L siempre tiene tamaño p; para los procesadores que no realizaron escrituras en el paso, L contiene direcciones inválidas (por ejemplo, -1).

Nuestro simulador recibe la lista L y debe decidir cómo realizar las escrituras. Como tenemos p procesadores, podemos asignar uno a cada escritura; sin embargo, como queremos pasar al modelo CREW, sólo podemos permitir una escritura por ubicación. Lo que hacemos, entonces, es ordenar el arreglo L por ubicaciones: cuando hagamos esto, se crearán "bloques" de elementos de L. Por ejemplo, L podría verse así:

$$
\begin{gathered}
(1,9)\\
(1,9)\\
(1,9)\\
(1,9)\\
(2,6)\\
(2,6)\\
(2,6)\\
\vdots
\end{gathered}
$$

Sólo tenemos que realizar una escritura por cada uno de estos bloques (es decir, escribir 9 en la ubicación 1 y 6 en la ubicación 2 sólo una vez). Considerando que tenemos procesadores $P_1 \dots P_p$, el procesador P_i $i > 1$ observa $L[i] = (l_i, v_i)$ y $L[i-1] = (l_{i-1}, v_{i-1})$:

- Si $l_i = -1$, entonces el procesador que P_i está intentando simular no realizó una escritura en este paso, así que no hay nada que hacer.
- Si $l_i \neq -1$ y $l_i \neq li - 1$, significa que P_i tiene asignado el inicio de un bloque. En este caso P_i escribe v_i en la ubicación l_i.
- Si $l_i \neq -1$ y $l_i = li - 1$, significa que P_i tiene asignado un punto en la mitad de un bloque. La escritura que P_i querría realizar ya fue manejada por el procesador asignado al inicio de ese bloque, así que no hay nada que hacer.

El tercer punto funciona porque el modelo que se nos dio impone que todas las escrituras en una misma ubicación tienen que escribir el mismo valor, por lo que no importa qué procesador de cada bloque haga la escritura (mientras sea sólo uno). El caso para $i = 1$ es más simple: sólo chequeamos si $l_1 \neq -1$: si l_1 es una ubicación válida escribimos v_1 allí, pues por definición es el primer elemento de su bloque.

¿Cuánto cuesta esto? Es necesario ordenar el arreglo L, lo que puede hacerse en tiempo $\log^2 p$, como se vio en clases.

P6. Sabemos que podemos computar *parallel prefix* (es decir, sumas acumuladas) de forma rápida. Entonces, una posible forma de resolver este problema es interpretar P como las sumas

acumuladas para algún operador binario asociativo (que tendremos que definir). Para no lanzarles la solución de forma "mágica", intentemos ver qué propiedades debería cumplir este operador.

Primero, las sumas acumuladas que sabemos calcular toman la forma $x_1, x_1 + x_2, x_1 + x_2 + x_3 \ldots$; sin embargo, las sumas que nos piden calcular no parten desde el primer término de la lista: parten desde $s(i)$, el último 1 visto. Esto significa que tenemos que poder "resetear" las sumas acumuladas, para descartar los términos que no queremos. Si el operador que queremos definir es $\hat{+}$, una situación en que las sumas se resetean es si se cumple $x \hat{+} y = y$ en ciertos casos. Esto causará que en las sumas acumuladas desechen todos los términos sumados hasta ese punto, comenzando de nuevo con y. Ahora, queremos que este "reset" pase cuando llegamos al último 1 (de modo que la suma final comience desde $s(i)$): para esto, es suficiente resetear la suma cada vez que vemos un 1.

Con esto en mente, definimos la operación $\hat{+}$ en $\{0,1\} \times D$ como sigue:

$$(a, z)\hat{+}(a', z') = \begin{cases} (1, z') & \text{si } a' = 1 \\ (a, z + z') & \text{si } a' = 0 \end{cases}$$

Esto cumple exactamente lo que queríamos: al calcular las sumas acumuladas, cada vez que veamos un 1 en B, la suma se va a resetear, pues se cae en el primer caso de la definición (y se desecha el término a la izquierda del operador, que contiene todo lo sumado hasta el momento). En otras palabras, si X es la suma acumulada de $B \times L$ con respecto a la operación $\hat{+}$, las segundas componentes de X corresponden a P :).

Adicionalmente, la operación es asociativa, lo que no es difícil de mostrar. Dados tres pares (a, x), (b, y), (c, z), estrictamente habría 8 casos que analizar para demostrar que es asociativa (ver todas las combinaciones de $(a, b, c) \in \{0, 1\}^3$). Sin embargo, de estos casos, aquellos en que $c = 1$ son triviales, ya que al tener un 1 en el operando de la derecha, el resultado de $(a, x)\hat{+}(b, y)\hat{+}(c, z)$ siempre será (c, z) (pues se aplica el primer caso de la definición). Adicionalmente, el caso $a = b = c = 0$ también es trivial, pues se basa en la asociatividad misma de $+$ (mientras en la primera componente se propaga el valor del operando de más a la izquierda). Nos faltan 3 de los 8 casos, en los que simplemente aplicamos la definición de $\hat{+}$:

Si $a = b = 1, c = 0$:

$$\begin{aligned} ((1, x)\hat{+}(1, y))\hat{+}(0, z) &\stackrel{?}{=} i(1, x)\hat{+}((1, y)\hat{+}(0, z)) \\ \iff (1, y)\hat{+}(0, z) &\stackrel{?}{=} (1, x)\hat{+}(1, y + z) \\ \iff (1, y + z) &= (1, y + z) \end{aligned}$$

Si $a = 1, b = 0, c = 0$:

$$\begin{aligned} ((1, x)\hat{+}(0, y))\hat{+}(0, z) &\stackrel{?}{=} (1, x)\hat{+}((0, y)\hat{+}(0, z)) \\ \iff (1, x + y)\hat{+}(0, z) &\stackrel{?}{=} (1, x)\hat{+}(0, y + z) \\ \iff (1, x + y + z) &= (1, x + y + z) \end{aligned}$$

Finalmente, si $a = 0, b = 1, c = 0$:

$$\begin{aligned} ((0, x)\hat{+}(1, y))\hat{+}(0, z) &\stackrel{?}{=} (0, x)\hat{+}((1, y)\hat{+}(0, z)) \\ \iff (1, y)\hat{+}(0, z) &\stackrel{?}{=} (0, x)\hat{+}(1, y + z) \\ \iff (1, y + z) &= (1, y + z) \end{aligned}$$

Como la operación es asociativa, podemos aplicar los algoritmos para *parallel prefix*: el algoritmo para calcular P consiste, en tonces en computar X, el arreglo con las sumas acumuladas de $B \times L$ con respecto a la operación $\hat{+}$ (no es necesario calcular explícitamente $B \times L$), guardando las segundas componentes de cada elemento en P. Esto puede hacerse, como se vio en clases, en tiempo $O(\log n)$ usando $O(\frac{n}{\log n})$ procesadores. La eficiencia es, finalmente, $\Theta(1)$.