Programando en paralelo

Mauro Jaskelioff

23/05/2022

Mergesort

- El algoritmo de ordenación mergesort es un clásico ejemplo de Divide & Conquer
- Dividimos la entrada en dos (*split*)
- Ordenamos recursivamente
- Juntamos las dos mitades ordenadas (*merge*)

Ordenando listas con Mergesort

$$
\begin{array}{lcl}
msort & : & [Int] \to [Int] \\
msort\ [\,] & = & [\,] \\
msort\ [x] & = & [x] \\
msort\ xs & = & \textbf{let}\ \begin{array}[t]{lcl} (ls, rs) & = & split\ xs \\ (ls', rs') & = & (msort\ ls \mid\mid msort\ rs) \end{array} \\
 & & \textbf{in}\ merge\ (ls', rs')
\end{array}
$$

$$
\begin{array}{lcl}
split & : & [Int] \to [Int] \times [Int] \\
split\ [\,] & = & ([\,], [\,]) \\
split\ [x] & = & ([x], [\,]) \\
split\ (x \lhd y \lhd zs) & = & \textbf{let}\ (xs, ys) = split\ zs \\
 & & \textbf{in}\ (x \lhd xs, y \lhd ys)
\end{array}
$$

Mergesort (cont.)

$$
\begin{aligned}
&\mathit{merge} && : [\mathit{Int}] \times [\mathit{Int}] \to [\mathit{Int}] \\
&\mathit{merge}\ ([\,], \mathit{ys}) && = \mathit{ys} \\
&\mathit{merge}\ (\mathit{xs}, [\,]) && = \mathit{xs} \\
&\mathit{merge}\ (x \triangleleft \mathit{xs}, y \triangleleft \mathit{ys}) && = \textbf{if}\ x \leqslant y\ \textbf{then}\ x \triangleleft \mathit{merge}\ (\mathit{xs}, y \triangleleft \mathit{ys}) \\
& && \phantom{= \textbf{if}\ x \leqslant y\ }\textbf{else}\ \ y \triangleleft \mathit{merge}\ (x \triangleleft \mathit{xs}, \mathit{ys})
\end{aligned}
$$

Mergesort: Trabajo

- $W_{split}(n) \in O(n)$
- $W_{merge}(n) \in O(n)$
- $W_{msort}(n) = c_0 n + 2W_{msort}(\frac{n}{2}) + c_1 n + c_2$
- Por lo tanto
$$W_{msort}(n) \in O(n \lg n)$$

Mergesort: Profundidad

- $S_{split}(n) \in O(n)$
- $S_{merge}(n) \in O(n)$
- $S_{msort}(n) = k_0 n + S_{msort}(\frac{n}{2}) + k_1 n + k_2$
- Por lo tanto,
$$S_{msort} \in O(n)$$
- ¡No es muy paralelizable!
- ¿Cuál es el problema?

Paralelizando Mergesort

- El problema no es el algoritmo, sino las listas
- *split* y *merge* son poco paralelizables.
- En general, las listas no son buenas para paralelizar

  La elección de la estructura de datos influye en la profundidad del algoritmo

- En lugar de listas trabajemos con el siguiente tipo de árboles:

$$\textbf{data}\ BT\ a = Empty \mid Node\ (BT\ a)\ a\ (BT\ a)$$

- ¡Podemos implementar *msort* sobre árboles con $W(n) \in O(n \lg n)$ y $S(n) \in O((\lg n)^3)$!

Árboles de búsqueda

- En elemento de $BT\ a$ está ordenado sii
  1. Es el árbol *Empty*
  2. Es un $Node\ l\ x\ r$ y ademas,
     - l está ordenado
     - r está ordenado
     - todos los elementos en l son $\leqslant x$
     - $x <$ todo elemento de r
  3. Un árbol ordenado induce una lista ordenada

$$
\begin{aligned}
&listar && :\ BT\ a \to [a] \\
&listar\ Empty && = [\,] \\
&listar\ (Node\ l\ x\ r) && = listar\ l +\!\!+ [x] +\!\!+ listar\ r
\end{aligned}
$$

     - Es un recorrido *inorder*

Mergesort sobre árboles

- ¿Qué pinta tendrá el *msort* sobre árboles?

$$\begin{array}{ll}
msort & : BT\ a \rightarrow BT\ a \\
msort\ Empty & = Empty \\
msort\ (Node\ l\ x\ r) & = \ \ldots\ msort\ l\ \ldots\ msort\ r\ \ldots
\end{array}$$

- Primer ventaja: $W_{split} \in O(1)$, $S_{split} \in O(1)$
- Hacemos un *merge* de *msort* l, *msort* r, y de x.

$$\begin{array}{ll}
msort & : BT\ a \rightarrow BT\ a \\
msort\ Empty & = Empty \\
msort\ (Node\ l\ x\ r) & = \mathbf{let}\ (l', r') = msort\ l \parallel msort\ r \\
 & \quad \mathbf{in}\ merge\ (merge\ l'\ r') \\
 & \qquad\qquad (Node\ Empty\ x\ Empty)
\end{array}$$

- Queremos que $W_{merge} \in O(n)$ y S_{merge} mejor que $O(n)$.

Merge de árboles (i)

- ¿Cómo definir *merge* sobre árboles?
- Consideremos el siguiente caso

$$\mathit{merge}\left(\begin{array}{ccc} & 3 & \\ 1 & & 5\end{array}, \begin{array}{ccc} & 4 & \\ 2 & & 6\end{array}\right)$$

- Elegimos guiarnos por el primer argumento.

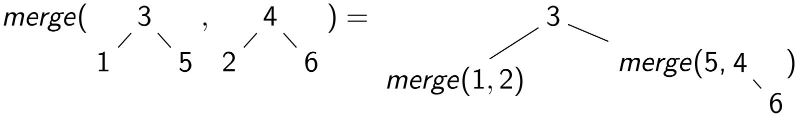
$$\mathit{merge}\left(\begin{array}{ccc} & 3 & \\ 1 & & 5\end{array}, \begin{array}{ccc} & 4 & \\ 2 & & 6\end{array}\right) = \begin{array}{ccc} & 3 & \\ \mathit{merge}(?,?) & & \mathit{merge}(?,?)\end{array}$$

- El 1 seguro va a la izquierda, el 5 a la derecha

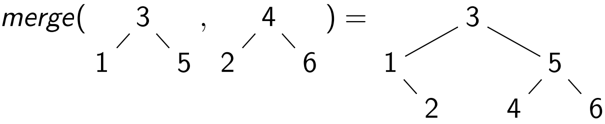
$$\mathit{merge}\left(\begin{array}{ccc} & 3 & \\ 1 & & 5\end{array}, \begin{array}{ccc} & 4 & \\ 2 & & 6\end{array}\right) = \begin{array}{ccc} & 3 & \\ \mathit{merge}(1,?) & & \mathit{merge}(5,?)\end{array}$$

Merge de árboles (ii)

- Separamos el segundo argumento en árboles menores a 3 y mayores a 3



- Finalmente



Merge de árboles (ii)

- La implementación de *merge* es:

$$
\begin{array}{lcl}
merge & : & BT\ Int \to BT\ Int \to BT\ Int \\
merge\ Empty\ t_2 & = & t_2 \\
merge\ (Node\ l_1\ x\ r_1)\ t_2 & = & \mathbf{let}\ (l_2, r_2) = splitAt\ t_2\ x \\
& & \qquad (l', r') = \begin{array}{l} merge\ l_1\ l_2 \\ \quad || \\ merge\ r_1\ r_2 \end{array} \\
& & \mathbf{in}\ Node\ l'\ x\ r'
\end{array}
$$

- donde *splitAt* se define:

$$
\begin{array}{lcl}
splitAt & : & BT\ Int \to Int \to Bt\ Int \times BT\ Int \\
splitAt\ Empty\ \_ & = & (Empty, Empty) \\
splitAt\ (Node\ l\ x\ r)\ y & = & \mathbf{if}\ y < x\ \mathbf{then}\ \mathbf{let}\ (ll, lr) = splitAt\ l\ y \\
& & \qquad\qquad\quad \mathbf{in}\ (ll, Node\ lr\ x\ r) \\
& & \qquad\qquad\ \mathbf{else}\ \ \mathbf{let}\ (rl, rr) = splitAt\ r\ y \\
& & \qquad\qquad\quad \mathbf{in}\ (Node\ l\ x\ rl, rr)
\end{array}
$$

Profundidad de *merge*

- Sea h la altura del árbol.
- $S_{splitAt}(h) = k + S_{SplitAt}(h-1) \quad \Rightarrow \quad S_{SplitAt}(h) \in O(h)$.
- Sean h_1 y h_2 las alturas de los árboles argumento

$$S_{merge}(h_1, h_2) \leqslant k + S_{SplitAt}(h_2) + \max(S_{merge}(h_1 - 1, h_{21}), \\ S_{merge}(h_1 - 1, h_{22}))$$

  donde h_{21} y h_{22} son las alturas de los árboles devueltos por *splitAt*

- $h_{21} \leqslant h_2, \quad h_{22} \leqslant h_2$

$$S_{merge}(h_1, h_2) \leqslant k + S_{SplitAt}(h_2) + \max(S_{merge}(h_1 - 1, h_2), \\ S_{merge}(h_1 - 1, h_2))$$

Profundidad de *merge* (cont.)

- Continuamos aproximando…

$$S_{merge}(h_1, h_2) \leqslant k + S_{SplitAt}(h_2) + \max(S_{merge}(h_1 - 1, h_2), S_{merge}(h_1 - 1, h_2))$$

$$S_{merge}(h_1, h_2) \leqslant k' h_2 + S_{merge}(h_1 - 1, h_2)$$

- Sumamos h_1 veces $(k' h_2)$
- Por lo tanto, $S_{merge}(h_1, h_2) \in O(h_1 \cdot h_2)$
- Si n es el tamaño del árbol, y el árbol está balanceado, entonces $h \in O(\lg n)$.

Profundidad de *msort*

- Calculamos la profundidad de *msort*

$$S_{msort}(n) \leqslant k + \max(S_{msort}(\frac{n}{2}), S_{msort}(\frac{n}{2})) + \\ S_{merge}(\lg n, \lg n) + S_{merge}(2\lg n, 1)$$

- El $(2\lg n)$ es porque $altura\ (merge\ l\ r) \leqslant altura\ l + altura\ r$
- Como $S_{merge}(h_1, h_2) \in O(h_1 \cdot h_2)$

$$S_{msort}(n) \leqslant \ k + S_{msort}(\frac{n}{2}) + k_1(\lg n)^2 + k_2 \lg n$$

- Simplificando

$$S_{msort}(n) \leqslant \ k + S_{msort}(\frac{n}{2}) + k_3(\lg n)^2$$

- Por lo tanto

$$\boxed{S_{msort}(n) \in O((\lg n)^3)}$$

Mentira!

- El análisis de la profundidad tiene un error grave.
- La profundidad de *merge* suponía árboles balanceados
- ¡Pero en *msort* llamamos a *merge* con el resultado de las llamadas recursivas!
- Lo arreglamos con una función $\mathit{rebalance} :: \mathit{BT}\ a \to \mathit{BT}\ a$

$$
\begin{array}{lcl}
\mathit{msort} & : & \mathit{BT}\ a \to \mathit{BT}\ a \\
\mathit{msort}\ \mathit{Empty} & = & \mathit{Empty} \\
\mathit{msort}\ (\mathit{Node}\ l\ x\ r) & = & \mathbf{let}\ (l', r') = \mathit{msort}\ l \mathbin{||} \mathit{msort}\ r \\
 & & \mathbf{in}\ \mathit{rebalance}\ (\mathit{merge}\ (\mathit{merge}\ l'\ r') \\
 & & \qquad\qquad\qquad\qquad\quad (\mathit{Node}\ \mathit{Empty}\ x\ \mathit{Empty}) \\
 & & \qquad\qquad\quad)
\end{array}
$$

Comentarios

- Este algoritmo paralelo trabaja sobre árboles,
- pero la entrada podría ser una lista.
- Convertir una estructura secuencial en paralela puede no ser paralelizable.
- Por lo tanto no podríamos esperar una mejora lineal en la cant. de procesadores.

Programando con Árboles

- Veamos operaciones sobre los siguientes árboles

$$\textbf{data}\ T\ a = \mathit{Empty} \mid \mathit{Leaf}\ a \mid \mathit{Node}\ (T\ a)\ (T\ a)$$

- Map

$$\begin{array}{ll}
\mathit{mapT} & : (a \to b) \to T\ a \to T\ b \\
\mathit{mapT}\ f\ \mathit{Empty} & = \mathit{Empty} \\
\mathit{mapT}\ f\ (\mathit{Leaf}\ x) & = \mathit{Leaf}\ (f\ x) \\
\mathit{mapT}\ f\ (\mathit{Node}\ l\ r) & = \textbf{let}\ (l', r') = \mathit{mapT}\ f\ l \mid\mid \mathit{mapT}\ f\ r \\
& \ \textbf{in}\ \mathit{Node}\ l'\ r'
\end{array}$$

- Si suponemos que $W_f \in O(1)$ y $S_f \in O(1)$
  - $W_{(\mathit{mapT}\ f)} \in O(n)$
  - $S_{(\mathit{mapT}\ f)} \in O(\lg n)$

Otras funciones

- Sumar todos los elementos de un árbol de enteros

$$
\begin{aligned}
&\mathit{sumT} && : T\ \mathit{Int} \rightarrow \mathit{Int} \\
&\mathit{sumT}\ \mathit{Empty} && = 0 \\
&\mathit{sumT}\ (\mathit{Leaf}\ x) && = x \\
&\mathit{sumT}\ (\mathit{Node}\ l\ r) && = \mathbf{let}\ (l', r') = \mathit{sumT}\ l \mid\mid \mathit{sumT}\ r \\
& && \ \mathbf{in}\ l' + r'
\end{aligned}
$$

- Aplanar un árbol de cadenas

$$
\begin{aligned}
&\mathit{flattenT} && : T\ \mathit{String} \rightarrow \mathit{String} \\
&\mathit{flattenT}\ \mathit{Empty} && = [\,] \\
&\mathit{flattenT}\ (\mathit{Leaf}\ xs) && = xs \\
&\mathit{flattenT}\ (\mathit{Node}\ l\ r) && = \mathbf{let}\ (l', r') = \mathit{flattenT}\ l \mid\mid \mathit{flattenT}\ r \\
& && \ \mathbf{in}\ l' +\!\!+\ r'
\end{aligned}
$$

Reduce

- Estas funciones se pueden escribir como un *reduceT*

$$
\begin{aligned}
&\mathit{reduceT} &&: (a \to a \to a) \to a \to T\ a \to a \\
&\mathit{reduceT}\ f\ e\ \mathit{Empty} &&= e \\
&\mathit{reduceT}\ f\ e\ (\mathit{Leaf}\ x) &&= x \\
&\mathit{reduceT}\ f\ e\ (\mathit{Node}\ l\ r) &&= \mathbf{let}\ (l', r') = \mathit{reduceT}\ f\ e\ l \\
& && \qquad\qquad\qquad\quad \| \\
& && \qquad\qquad\qquad\quad \mathit{reduceT}\ f\ e\ r \\
& && \quad\ \mathbf{in}\ f\ l'\ r'
\end{aligned}
$$

- $\mathit{sumT} = \mathit{reduceT}\ (+)\ 0$
- $\mathit{flattenT} = \mathit{reduceT}\ (+\!\!+)\ [\,]$
- Si $f \in O(1)$, entonces $W_{\mathit{reduce}}(n) \in O(n)$
- Si $f \in O(1)$, entonces $S_{\mathit{reduce}}(n) \in O(\lg n)$

Ejemplos

- Queremos saber la longitud (en palabras) de la línea con más palabras en un texto.
  - $lolile : String \to Int$
- Si tenemos una función $wordcount : String \to Int$, entonces

$$lolile = reduceT\ max\ 0\ .\ mapT\ wordcount\ .\ lines$$

  - $lines$ divide una cadena en un árbol de líneas

Contando palabras

- Contar palabras es igual de simple

$$\begin{aligned} \mathit{wordcount} &: \mathit{String} \to \mathit{Int} \\ \mathit{wordcount} &= \mathit{sumT}\ .\ \mathit{mapT}\ (\lambda\_ \to 1)\ .\ \mathit{words} \end{aligned}$$

- $\mathit{words} : \mathit{String} \to T\ \mathit{String}$, divide una cadena en un árbol de palabras.
- En resumen:

$$\begin{aligned} \mathit{lolile} &= \mathit{reduceT}\ \mathit{max}\ 0\ .\ \mathit{mapT}\ \mathit{wordcount}\ .\ \mathit{lines} \\ \mathit{wordcount} &= \mathit{reduceT}\ (+)\ \ 0\ .\ \mathit{mapT}\ (\lambda\_ \to 1)\ \ .\ \mathit{words} \end{aligned}$$

mapreduce

- Hacer un *mapT* y luego un *reduceT* es ineficiente
- *mapT* genera un árbol que es inmediatamente consumido por *reduceT*
- Las dos funciones se pueden combinar en una sola:

$$
\begin{array}{l}
mapreduce \qquad\quad : (a \to b) \to (b \to b \to b) \to b \to T\ a \to b \\
mapreduce\ f\ g\ e = mr \\
\quad \textbf{where}\ \begin{array}[t]{lcl} mr\ Empty & = & e \\ mr\ (Leaf\ a) & = & f\ a \\ mr\ (Node\ l\ r) & = & \textbf{let}\ (l', r') = mr\ l\ ||\ mr\ r \\ & & \textbf{in}\ g\ l'\ r' \end{array}
\end{array}
$$

- *mapreduce* nos da otro ejemplo del uso del alto orden para expresar patrones de programación como programas.

Resumen

- En general, las listas no son muy paralelizables.
- Para paralelizar, conviene trabajar con otras estructuras, como por ejemplo árboles.
- Las funciones de alto orden nos permiten capturar patrones generales de recursión.