DECSAI
Departamento de Ciencias de la Computación e I.A.
Universidad de Granada

Algoritmos paralelos de ordenación
Análisis y Diseño de Algoritmos

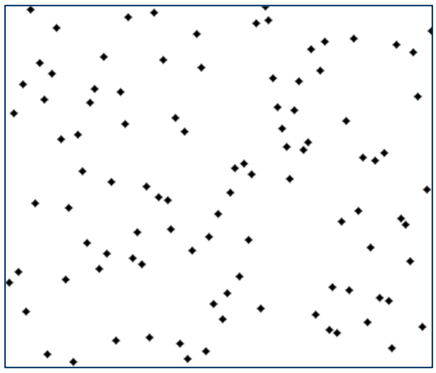
Algoritmos paralelos de ordenación

- Recordatorio: MergeSort
- MergeSort paralelo
  - Paralelización de las llamadas recursivas
  - Mezcla paralela (binaria)
  - Mezcla paralela multivía (de k vías)
  - El algoritmo más rápido: TencentSort (2016)
- QuickSort paralelo
  - Suma de prefijos
  - Partición paralela

MergeSort

Algoritmo de ordenación "divide y vencerás":

1. Dividir nuestro conjunto en dos mitades.
2. Ordenar recursivamente cada mitad.
3. Combinar las dos mitades ordenadas: O(n).



MergeSort

```
algorithm mergesort(A, lo, hi):
    if lo+1 < hi:                    // Two or more elements.
        mid = ⌊(lo + hi) / 2⌋
        mergesort(A, lo, mid)
        mergesort(A, mid, hi)
        merge(A, lo, mid, hi)
```

A	L	G	O	R	I	T	H	M	S

A	L	G	O	R		I	T	H	M	S	Dividir	O(1)

A	G	L	O	R		H	I	M	S	T	Ordenar	2T(n/2)

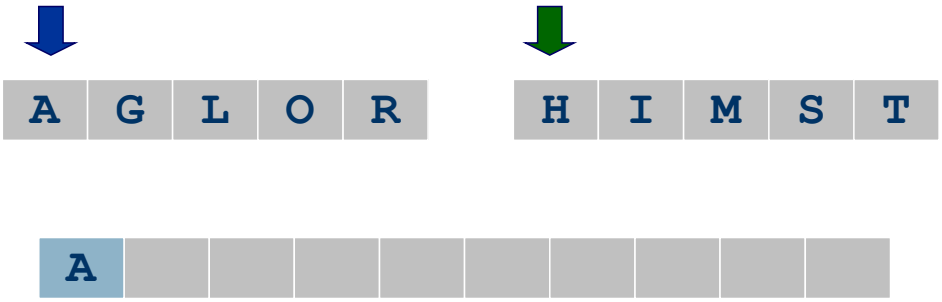
A	G	H	I	L	M	O	R	S	T	Mezclar	O(n)

MergeSort

¿Cómo combinar eficientemente dos listas ordenadas?

Usando un array auxiliar y un número lineal de comparaciones:

- Controlar la posición del elemento más pequeño en cada mitad.
- Añadir el más pequeño de los dos a un vector auxiliar.
- Repetir hasta que se hayan añadido todos los elementos.



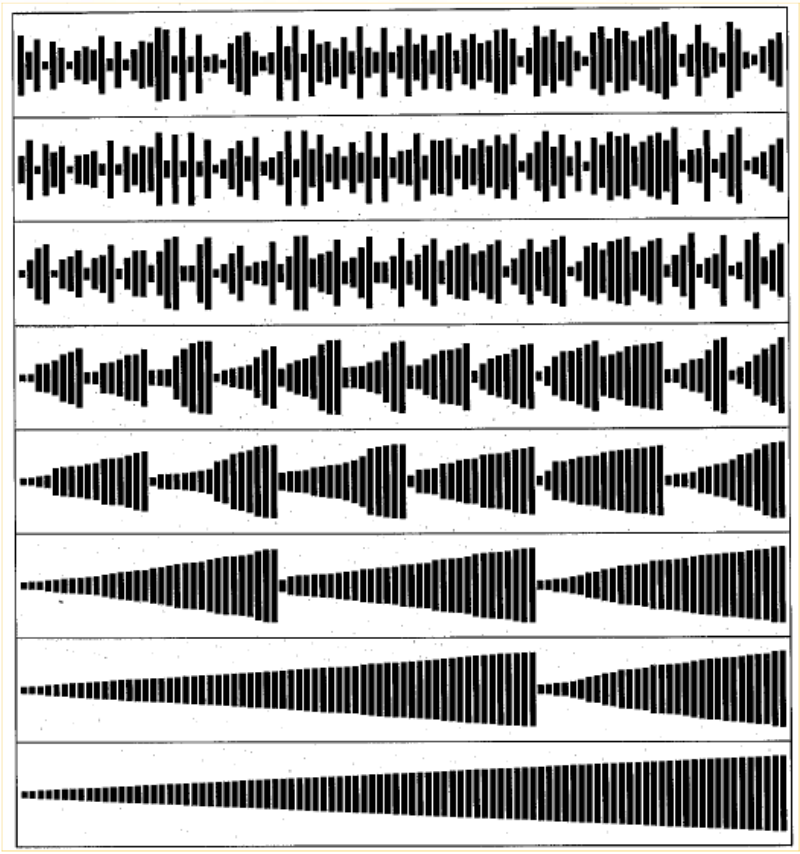
Optimización: "In-place merge" (Kronrud, 1969)

Cantidad constante de almacenamiento extra.

MergeSort

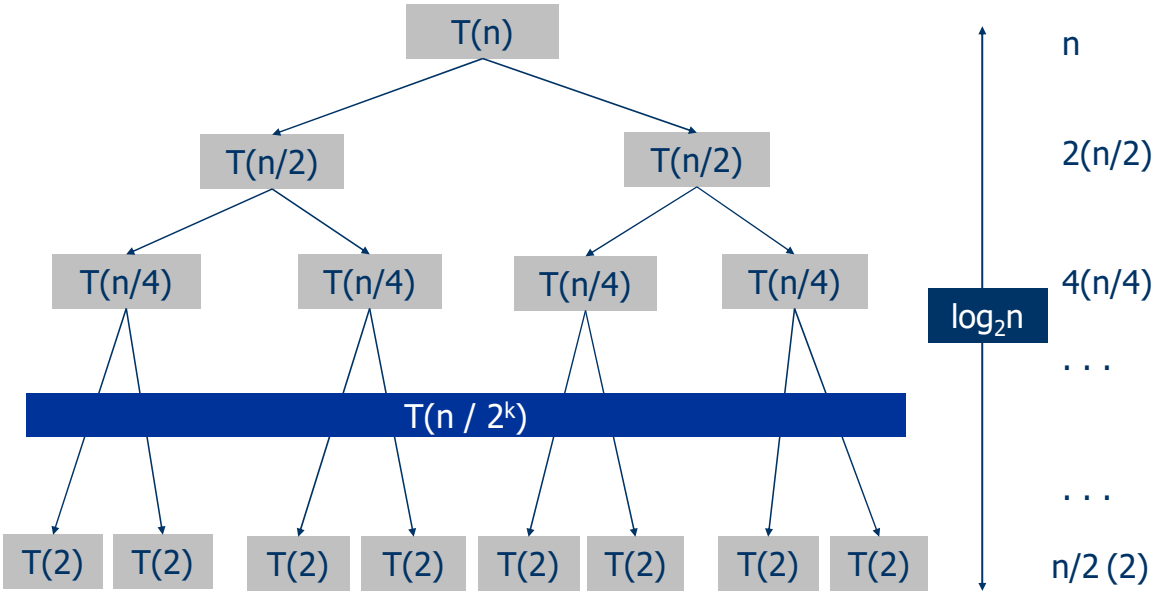
```
algorithm merge(A, B):

    C = new empty list
    while A is not empty and B is not empty:
        if head(A) ≤ head(B):
            append head(A) to C & drop the head of A
        else:
            append head(B) to C & drop the head of B

    // Either A or B is empty, just empty the other input list.
    while A is not empty:
        append head(A) to C & drop the head of A
    while B is not empty:
        append head(B) to C & drop the head of B

    return C
```

MergeSort



MergeSort: Eficiencia

$$T(n) = \begin{cases} 1, & n = 1 \\ 2T(n/2) + n, & n > 1 \end{cases}$$



MergeSort: Eficiencia

$$T(n) = \begin{cases} 1, & n = 1 \\ 2T(n/2) + n, & n > 1 \end{cases}$$

Expandiendo la recurrencia:

$$\begin{aligned} \frac{T(n)}{n} &= \frac{2T(n/2)}{n} && +1 \\ &= \frac{T(n/2)}{n/2} && +1 \\ &= \frac{T(n/4)}{n/4} && +1+1 \\ &\cdots \\ &= \frac{T(n/n)}{n/n} && \underbrace{+1+\cdots+1}_{\log_2 n} \\ &= \log_2 n \end{aligned}$$

MergeSort: Eficiencia

$$T(n) = \begin{cases} 1, & n = 1 \\ 2T(n/2) + n, & n > 1 \end{cases}$$

Usando el método de la ecuación característica:

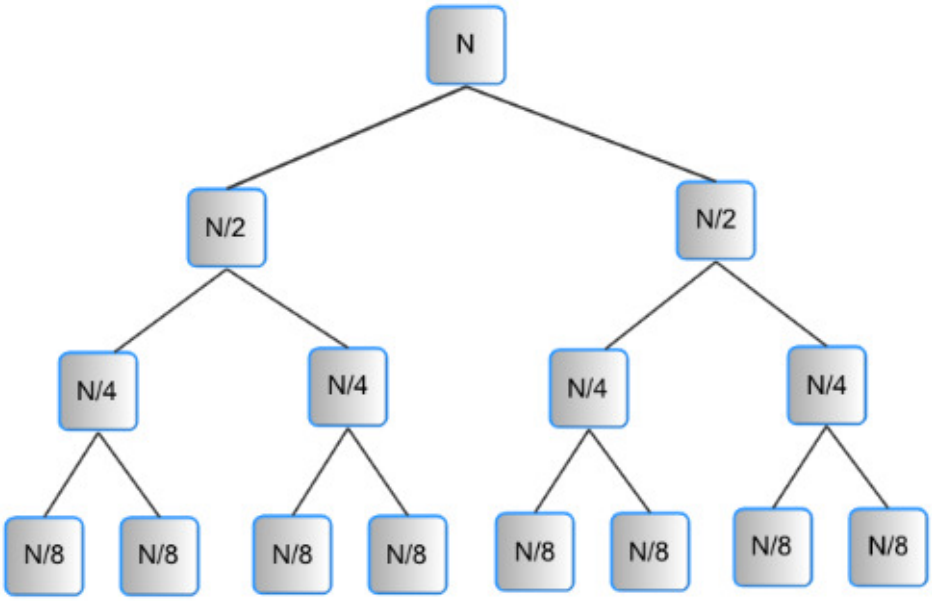
$$t_i = T(2^i) \qquad t_i = 2t_{i-1} + 2^i \qquad \Rightarrow \qquad t_i = c_1 2^i + c_2 i 2^i$$

$$T(n) = c_1 2^{\log_2(n)} + c_2 \log_2(n) 2^{\log_2(n)} = c_1 n + c_2 n \log_2(n)$$

$$T(n) \text{ es } \mathrm{O}(n \log_2 n)$$

MergeSort paralelo

Paralelización de la llamada recursiva



MergeSort paralelo

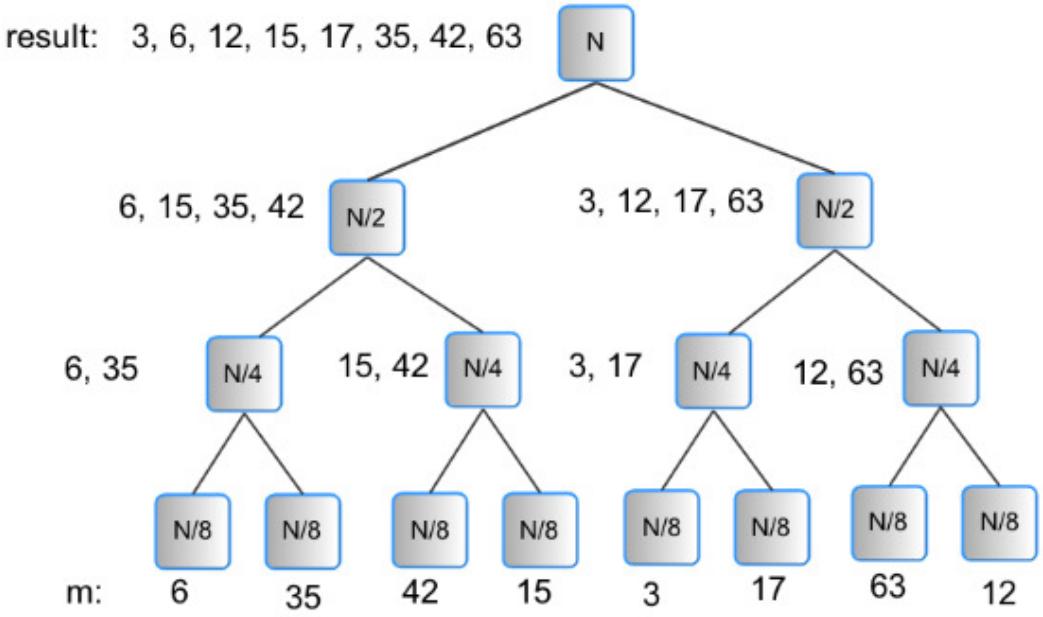
Paralelización de la llamada recursiva

```
algorithm mergesort(A, lo, hi):
    if lo+1 < hi:  // Two or more elements.
        mid = ⌊(lo + hi) / 2⌋
        fork mergesort(A, lo, mid)
        mergesort(A, mid, hi)
        join
        merge(A, lo, mid, hi)
```

Modificación trivial del algoritmo secuencial,
no paraleliza bien: tiempo $\Theta(n)$,
mejora [speedup] sólo de $\Theta(\log n)$,
por la ejecución secuencial de la mezcla.

MergeSort paralelo

Mezcla paralela



MergeSort paralelo

Mezcla paralela [Cormen et al.]

```
algorithm parallelMergesort(A, lo, hi, B, offset):
    len = hi - lo + 1
    if len == 1:
        B[offset] = A[lo]
    else:
        let T[1..len] be a new array
        mid = ⌊(lo + hi) / 2⌋
        mid' = mid - lo + 1
        fork parallelMergesort(A, lo, mid, T, 1)
        parallelMergesort(A, mid+1, hi, T, mid'+1)
        join
        parallelMerge(T, 1, mid', mid'+1, len, B, offset)
```

MergeSort paralelo

Mezcla paralela [Cormen et al.]

```
algorithm merge(A[i...j], B[k...ℓ], C[p...q]):
    m = j - i
    n = ℓ - k
    if m < n:
        swap A and B  // ensure that A is the larger array
        swap m and n
    if m > 0:         // m≤0 == base case, nothing to merge
        r = ⌊(i + j)/2⌋
        s = binary-search(A[r], B[k...ℓ])
        t = p + (r - i) + (s - k)
        C[t] = A[r]
        in parallel do
            merge(A[i...r], B[k...s], C[p...t])
            merge(A[r+1...j], B[s...ℓ], C[t+1...q])
```

MergeSort paralelo

- El algoritmo de mezcla paralela divide el más grande de A o B en dos mitades (que siempre estará en A).
- A continuación, divide el otro en dos partes con valores menores (parte izquierda) o mayores (parte derecha) que el punto medio de A, para lo que utiliza una búsqueda binaria, $\Theta(\log n)$.
- El trabajo realizado por el algoritmo paralelo de mezcla es, como en el algoritmo secuencial, O(n).
- Para obtener el tiempo de ejecución del algoritmo paralelo hemos de plantear una recurrencia…

MergeSort paralelo

- En el algoritmo paralelo de mezcla hay dos llamadas recursivas. Como ambas se ejecutan en paralelo, sólo hemos de considerar la más costosa.

- En el peor de los casos, el número de elementos en una de las llamadas recursivas será (3/4)n, ya que la parte más grande, ≥n/2, siempre se divide por la mitad.

- Teniendo en cuenta el coste de la búsqueda binaria:

$$T_{merge}(n) = T_{merge}\left(\frac{3}{4}n\right) + \Theta(\log n)$$

MergeSort paralelo

Tiempo necesario para la mezcla paralela
en una máquina ideal
(con un número ilimitado de procesadores).

$$T_{merge}(n) = T_{merge}\left(\frac{3}{4}n\right) + \Theta(\log n)$$

$$T_{merge}(n) \in \Theta((\log n)^2)$$

MergeSort paralelo

Tiempo necesario para el MergeSort paralelo en una máquina ideal (con un número ilimitado de procesadores).

$$T_{MergeSort}(n) = T_{MergeSort}\left(\frac{n}{2}\right) + T_{merge}(n)$$

$$T_{MergeSort}(n) = T_{MergeSort}\left(\frac{n}{2}\right) + \Theta((\log n)^2)$$

$$T_{MergeSort}(n) \in \Theta((\log n)^3)$$

MergeSort paralelo

Tiempo de ejecución en n procesadores:

$$T_{MergeSort}(n) \in \Theta((\log n)^3)$$

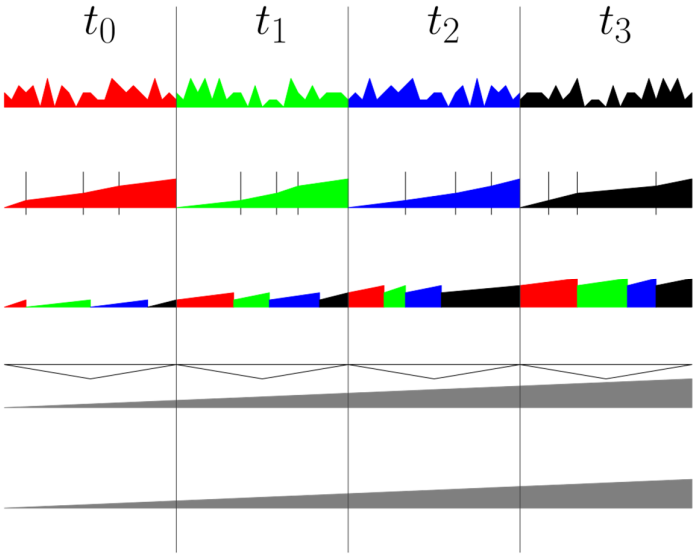
Mejora con respecto al algoritmo secuencial:

$$S_{MergeSort}(n) \in \Theta\left(\frac{n \log n}{(\log n)^3}\right) = \Theta\left(\frac{n}{(\log n)^2}\right)$$

Un algoritmo paralelo de este tipo funciona bien combinado con un algoritmo secuencial rápido (p.ej. inserción) y una implementación secuencial del MergeSort para vectores pequeños.

MergeSort paralelo

Mezcla paralela de k vías
[k-way / multiway MergeSort]



Generalización de la mezcla binaria, adecuada para sistemas paralelos

MergeSort paralelo

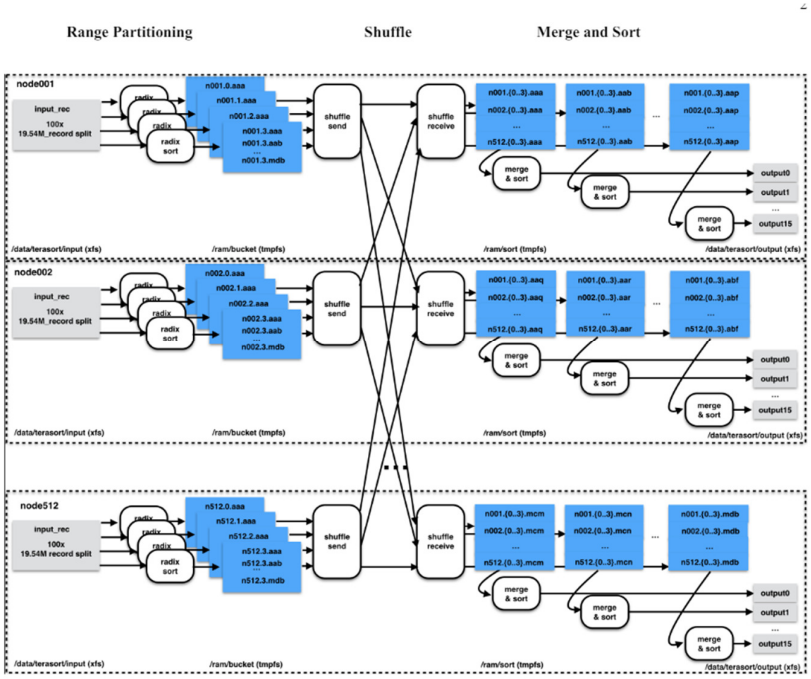
MergeSort con mezcla paralela de k vías
[k-way / multiway MergeSort]

$$O\left(\left(\frac{n}{p}+p\log n\right)\log\frac{n}{p}+\frac{n}{p}\log p\right)$$

- Cada procesador ordena n/p datos secuencialmente, $O\left(\frac{n}{p}\log\frac{n}{p}\right)$.
- Dadas p secuencias ordenadas, se reparten equitativamente en p partes: selección de puntos de corte (QuickSelect) con p búsquedas binarias, $O\left(p\log\frac{n}{p}\log n\right)$.
- Cada procesador mezcla las p listas que le corresponden [k-way merge]: $O\left(\frac{n}{p}\log p\right)$

TencentSort

Algoritmo más rápido para ordenar 100TB (según https://sortbenchmark.org/):



134 segundos

TencentSort

Fase 1: Particionamiento del rango

- RadixSort, a.k.a. bucket sort (algoritmo de ordenación que no requiere la realización de comparaciones).

Fase 2: Network shuffle

- Intercambio de datos entre nodos del cluster.

Fase 3: Merge & Sort

- Implementación paralela de MergeSort en cada nodo (usando multithreading).

Más info… @ https://sortbenchmark.org/TencentSort2016.pdf

QuickSort paralelo

```
algorithm quicksort (A, first, last):
   if last>first+1:
      select random p in [first:last]  // A[p] is pivot
      k = split(A, A[p]);
      quicksort(A, first, k-1)
      quicksort(A, k+1, last)
```

IDEA:
Si contamos cuantos valores hay en A menores que A[p] antes que A[i], sabemos cuál debe ser la posición de A[i] al particionar el vector A.

- La suma de prefijos [prefix sum] del vector X no es más que el vector S con S[i] = X[0] + X[1] +…+ X[i]
- Dada la suma de prefijos S, p.ej. A[S[p]] = A[p]

QuickSort paralelo

Suma de prefijos: Algoritmo secuencial

```
algorithm prefixsum (x):
   s[0] = x[0]
   for i=1..n-1:
      s[i] = s[i-1] + x[i]
   return s
```

- Tiempo de ejecución: O(n)

QuickSort paralelo

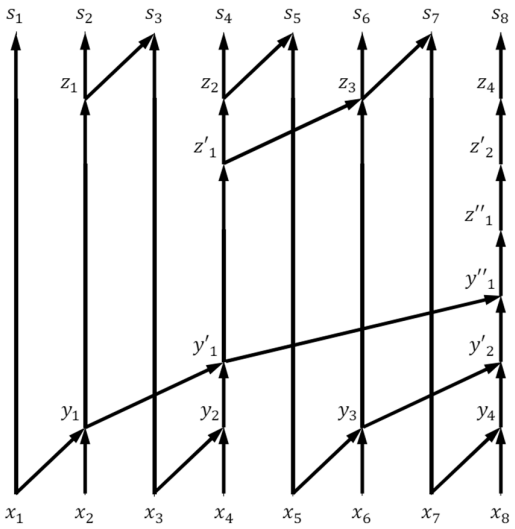
Suma de prefijos: Algoritmo paralelo

```
algorithm prefixsum (x):   // x vector de longitud n
   if n==1:
      s[0] = x[0]
   else:
      parallel for i=0..n/2-1:
         y[i] = x[2*i] + x[2*i+1]
      z = prefixsum(y)
      parallel for i=0..n-1:
         if i==0: s[0] = x[0]
         else if i is even: s[i] = z[i/2]
         else: s[i] = z[i/2-i] + x[i]
   return s
```


QuickSort paralelo

Suma de prefijos: Algoritmo paralelo

- Trabajo:
  $W(n) = W(n/2) + \Theta(n) \in \Theta(n)$
- Tiempo de ejecución:
  $T(n) = T(n/2) + \Theta(1) \in \Theta(\log n)$
- Paralelismo [speedup]:
  $S(n) \in \Theta(n / \log n)$



NOTA: En la práctica, el for paralelo se implementa usando "divide y vencerás" en $\Theta(\log n)$, luego $T(n) \in \Theta(\log^2 n)$

QuickSort paralelo

Partición paralela

Dado un vector A[q:r] y un elemento x de ese vector, reorganiza los elementos del vector de tal forma que todos los elementos menores que x estén en A[q:k-1] y todos los elementos mayores que x estén en A[k+1:r].

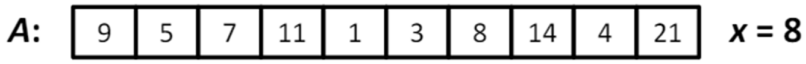
QuickSort paralelo

Partición paralela

```
algorithm split (A[q:r], x):
   n = q-r+1
   if n==1: return q
   else:
      parallel for i=0..n-1:
         B[i] = A[q+i]
         lt[i] = (B[i]<x)
         gt[i] = (B[i]>x)
      lt = prefixsum(lt)
      gt = prefixsum(gt)
      k = q + lt[n-1]; A[k] = x
      parallel for i=0..n-1:
         if B[i]<x: A[q+lt[i]-1] = B[i]
         else if B[i]>x: A[k+gt[i]] = B[i]
   return k
```

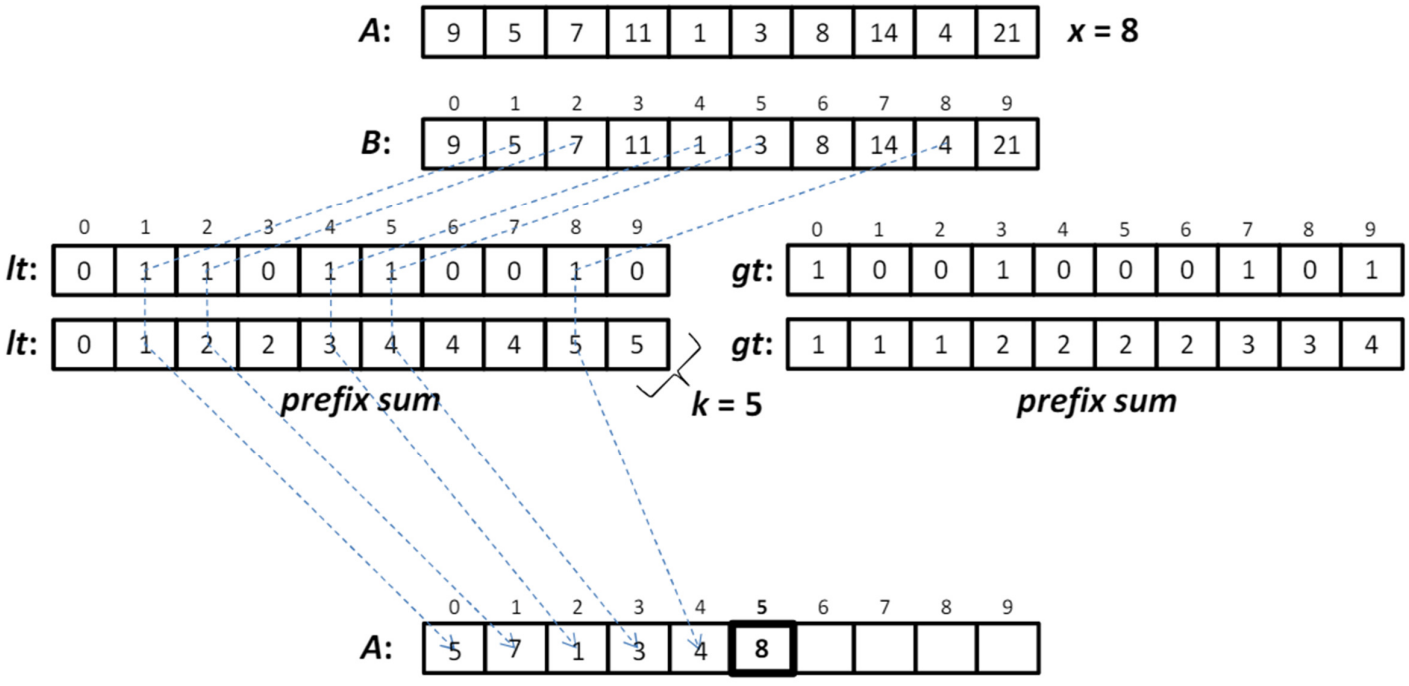
QuickSort paralelo

Partición paralela



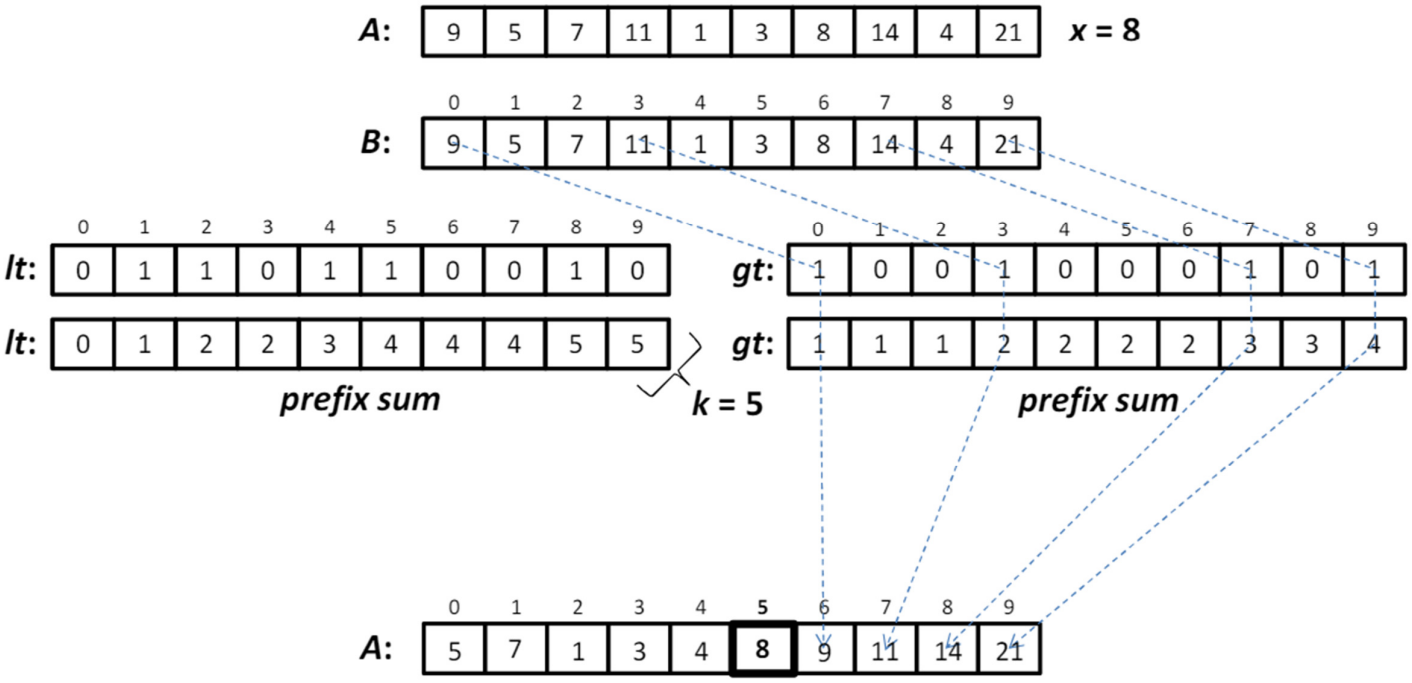
QuickSort paralelo

Partición paralela



QuickSort paralelo

Partición paralela



QuickSort paralelo

Partición paralela

- Trabajo:
  $W(n) \in \Theta(n)$

- Tiempo de ejecución:
  - $\Theta(1)$ parallel for (ideal):
    $T(n) = \Theta(1) + \Theta(\log n) \in \Theta(\log n)$
  - $\Theta(\log n)$ parallel for (realista):
    $T(n) = \Theta(\log n) + \Theta(\log^2 n) \in \Theta(\log^2 n)$

- Paralelismo [speedup]:
  $S(n) \in \Theta(n / \log^2 n)$

QuickSort paralelo

Algoritmo paralelo de ordenación

```
algorithm quicksort (A, first, last):
   n = last-first+1
   if n is small:
      sort(A,first,last)          // Sequential algorithm
   else:
      select random p in [first:last]  // A[p] is pivot
      k = split(A, A[p])
      fork quicksort(A, first, k-1)
      quicksort(A, k+1, last)
      join
```

QuickSort paralelo

Eficiencia del algoritmo paralelo

- Partición:
  $\Theta(\log^2 n)$ para realizar un trabajo $\Theta(n)$
- En cada nivel de la llamada recursiva, en el peor caso, el esfuerzo necesario para realizar la partición:
  Tiempo $O(\log^2 n)$ para realizar un trabajo $O(n)$.
- Si tenemos un árbol de llamadas recursivas de profundidad D: $T(n) \in O(D \log^2 n)$ y $W(n) \in O(D\ n)$
- Con una elevada probabilidad, $D \in O(\log n)$, por lo que **$T(n) \in O(\log^3 n)$** y **$W(n) \in O(n \log n)$**.