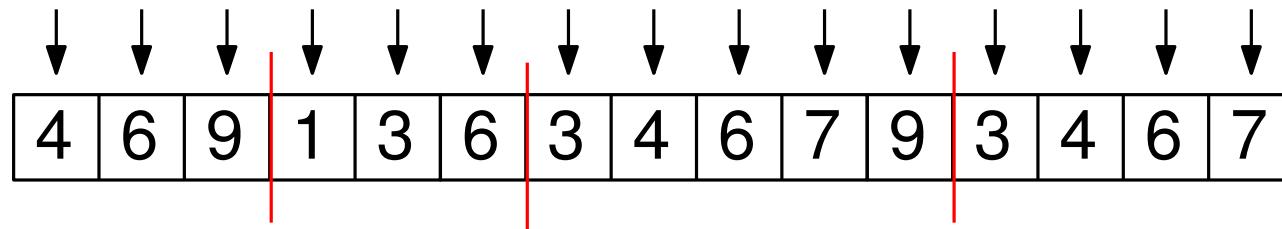
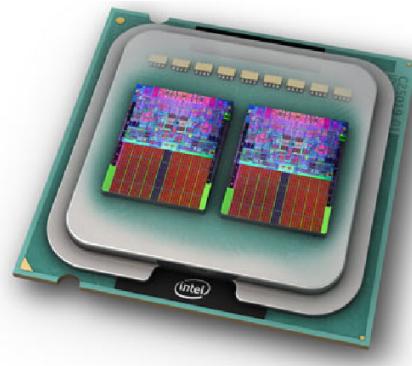




ICS 643: Parallel Algorithms

Prof. Nodari Sitchinava



Lecture 7: Segmented Prefix Sums II

Reminder: Segmented Prefix Sums

$b :$	1	0	0	1	0	0	1	0	0	0	0	1	0	0	0
$x :$	4	2	3	1	2	3	3	1	2	1	2	3	1	2	1
$y :$	4	6	9	1	3	6	3	4	6	7	9	3	4	6	7

$$y_i = \begin{cases} x_1 & \text{if } i = 1 \\ (y_{i-1} \cdot (1 - b_i)) + x_i & \text{if } i > 1 \end{cases}$$

Generalized Segmented Scan

$b :$	1	0	0	1	0	0	1	0	0	0	0	1	0	0	0
$x :$	4	2	3	1	2	3	3	1	2	1	2	3	1	2	1
$y :$	4	6	9	1	3	6	3	4	6	7	9	3	4	6	7

$$y_i = \begin{cases} x_1 & \text{if } i = 1 \\ (y_{i-1} \otimes b_i) \oplus x_i & \text{if } i > 1 \end{cases}$$

Generalized Segmented Scan

$b :$	1	0	0	1	0	0	1	0	0	0	0	1	0	0	0
$x :$	4	2	3	1	2	3	3	1	2	1	2	3	1	2	1
$y :$	4	6	9	1	3	6	3	4	6	7	9	3	4	6	7

$$y_i = \begin{cases} x_1 & \text{if } i = 1 \\ (y_{i-1} \otimes b_i) \oplus x_i & \text{if } i > 1 \end{cases}$$

$$\begin{pmatrix} x \\ b_1 \end{pmatrix} \bullet \begin{pmatrix} y \\ b_2 \end{pmatrix} = \begin{pmatrix} (x \otimes b_2) \oplus y \\ b_1 \vee b_2 \end{pmatrix}$$

Generalized Segmented Scan

$b :$	1	0	0	1	0	0	1	0	0	0	0	1	0	0	0
$x :$	4	2	3	1	2	3	3	1	2	1	2	3	1	2	1
$y :$	4	6	9	1	3	6	3	4	6	7	9	3	4	6	7

$$y_i = \begin{cases} x_1 & \text{if } i = 1 \\ (y_{i-1} \otimes b_i) \oplus x_i & \text{if } i > 1 \end{cases}$$

$$\begin{pmatrix} x \\ b_1 \end{pmatrix} \bullet \begin{pmatrix} y \\ b_2 \end{pmatrix} = \begin{pmatrix} (x \otimes b_2) \oplus y \\ b_1 \vee b_2 \end{pmatrix}$$

Scan

$$y_i = \begin{cases} x_1 & \text{if } i = 1 \\ y_{i-1} \oplus x_i & \text{if } i > 1 \end{cases}$$

Generalized Segmented Scan

$b :$	1	0	0	1	0	0	1	0	0	0	0	1	0	0	0
$x :$	4	2	3	1	2	3	3	1	2	1	2	3	1	2	1
$y :$	4	6	9	1	3	6	3	4	6	7	9	3	4	6	7

$$y_i = \begin{cases} x_1 & \text{if } i = 1 \\ (y_{i-1} \otimes b_i) \oplus x_i & \text{if } i > 1 \end{cases}$$

$$\begin{pmatrix} x \\ b_1 \end{pmatrix} \bullet \begin{pmatrix} y \\ b_2 \end{pmatrix} = \begin{pmatrix} (x \otimes b_2) \oplus y \\ b_1 \vee b_2 \end{pmatrix}$$

Scan

$$y_i = \begin{cases} x_1 & \text{if } i = 1 \\ y_{i-1} \oplus x_i & \text{if } i > 1 \end{cases}$$

$$\begin{pmatrix} y_i \\ b'_i \end{pmatrix} = \begin{cases} \begin{pmatrix} x_1 \\ b_1 \end{pmatrix} & \text{if } i = 1 \\ \begin{pmatrix} y_{i-1} \\ b'_{i-1} \end{pmatrix} \bullet \begin{pmatrix} x_i \\ b_i \end{pmatrix} & \text{if } i > 1 \end{cases}$$

Generalized Segmented Scan

$b :$	1	0	0	1	0	0	1	0	0	0	0	1	0	0	0
$x :$	4	2	3	1	2	3	3	1	2	1	2	3	1	2	1
$y :$	4	6	9	1	3	6	3	4	6	7	9	3	4	6	7

$$y_i = \begin{cases} x_1 & \text{if } i = 1 \\ (y_{i-1} \otimes b_i) \oplus x_i & \text{if } i > 1 \end{cases}$$

$$\begin{pmatrix} x \\ b_1 \end{pmatrix} \bullet \begin{pmatrix} y \\ b_2 \end{pmatrix} = \begin{pmatrix} (x \otimes b_2) \oplus y \\ b_1 \vee b_2 \end{pmatrix}$$

Scan

$$y_i = \begin{cases} x_1 & \text{if } i = 1 \\ y_{i-1} \oplus x_i & \text{if } i > 1 \end{cases}$$

$$\begin{pmatrix} y_i \\ b'_i \end{pmatrix} = \begin{cases} \begin{pmatrix} x_1 \\ b_1 \end{pmatrix} & \text{if } i = 1 \\ \begin{pmatrix} y_{i-1} \\ b'_{i-1} \end{pmatrix} \bullet \begin{pmatrix} x_i \\ b_i \end{pmatrix} = \begin{pmatrix} (y_{i-1} \otimes b_i) \oplus x_i \\ b'_{i-1} \vee b_i \end{pmatrix} & \text{if } i > 1 \end{cases}$$

Generalized Segmented Scan

$b :$	1	0	0	1	0	0	1	0	0	0	0	1	0	0	0
$x :$	4	2	3	1	2	3	3	1	2	1	2	3	1	2	1
$y :$	4	6	9	1	3	6	3	4	6	7	9	3	4	6	7

$$y_i = \begin{cases} x_1 & \text{if } i = 1 \\ (y_{i-1} \otimes b_i) \oplus x_i & \text{if } i > 1 \end{cases}$$

$$\begin{pmatrix} x \\ b_1 \end{pmatrix} \bullet \begin{pmatrix} y \\ b_2 \end{pmatrix} = \begin{pmatrix} (x \otimes b_2) \oplus y \\ b_1 \vee b_2 \end{pmatrix}$$

Scan

$$y_i = \begin{cases} x_1 & \text{if } i = 1 \\ y_{i-1} \oplus x_i & \text{if } i > 1 \end{cases}$$

$$\begin{pmatrix} y_i \\ b'_i \end{pmatrix} = \begin{cases} \begin{pmatrix} x_1 \\ b_1 \end{pmatrix} & \text{if } i = 1 \\ \begin{pmatrix} y_{i-1} \\ b'_{i-1} \end{pmatrix} \bullet \begin{pmatrix} x_i \\ b_i \end{pmatrix} = \begin{pmatrix} (y_{i-1} \otimes b_i) \oplus x_i \\ b'_{i-1} \vee b_i \end{pmatrix} & \text{if } i > 1 \end{cases}$$

Generalized Segmented Scan

$b :$	1	0	0	1	0	0	1	0	0	0	0	1	0	0	0
$x :$	4	2	3	1	2	3	3	1	2	1	2	3	1	2	1
$y :$	4	6	9	1	3	6	3	4	6	7	9	3	4	6	7

$$y_i = \begin{cases} x_1 & \text{if } i = 1 \\ (y_{i-1} \otimes b_i) \oplus x_i & \text{if } i > 1 \end{cases}$$

$$\begin{pmatrix} x \\ b_1 \end{pmatrix} \bullet \begin{pmatrix} y \\ b_2 \end{pmatrix} = \begin{pmatrix} (x \otimes b_2) \oplus y \\ b_1 \vee b_2 \end{pmatrix}$$

Scan

$$y_i = \begin{cases} x_1 & \text{if } i = 1 \\ y_{i-1} \oplus x_i & \text{if } i > 1 \end{cases}$$

Segmented scan

$$\begin{pmatrix} y_i \\ b'_i \end{pmatrix} = \begin{cases} \begin{pmatrix} x_1 \\ b_1 \end{pmatrix} & \text{if } i = 1 \\ \begin{pmatrix} y_{i-1} \\ b'_{i-1} \end{pmatrix} \bullet \begin{pmatrix} x_i \\ b_i \end{pmatrix} = \begin{pmatrix} (y_{i-1} \otimes b_i) \oplus x_i \\ b'_{i-1} \vee b_i \end{pmatrix} & \text{if } i > 1 \end{cases}$$

Segmented scan via prefix sums

Scan

$$y_i = \begin{cases} x_1 & \text{if } i = 1 \\ y_{i-1} \oplus x_i & \text{if } i > 1 \end{cases}$$

Segmented scan

$$\begin{pmatrix} y_i \\ b'_i \end{pmatrix} = \begin{cases} \begin{pmatrix} x_1 \\ b_1 \end{pmatrix} & \text{if } i = 1 \\ \begin{pmatrix} y_{i-1} \\ b'_{i-1} \end{pmatrix} \bullet \begin{pmatrix} x_i \\ b_i \end{pmatrix} & \text{if } i > 1 \end{cases}$$

Claim. • *is associative*

Segmented scan via prefix sums

Scan

$$y_i = \begin{cases} x_1 & \text{if } i = 1 \\ y_{i-1} \oplus x_i & \text{if } i > 1 \end{cases}$$

Segmented scan

$$\begin{pmatrix} y_i \\ b'_i \end{pmatrix} = \begin{cases} \begin{pmatrix} x_1 \\ b_1 \end{pmatrix} & \text{if } i = 1 \\ \begin{pmatrix} y_{i-1} \\ b'_{i-1} \end{pmatrix} \bullet \begin{pmatrix} x_i \\ b_i \end{pmatrix} & \text{if } i > 1 \end{cases}$$

Claim. • *is associative*

```
procedure PREFIX-SUMS( $x[1..n]$ )
  if  $n \leq 1$  then return
  for  $i = 1$  to  $\frac{n}{2}$  in parallel do
     $x'[i] = x[2i - 1] + x[2i]$ 
    PREFIX-SUMS( $x'[1..\frac{n}{2}]$ )
  for  $i = 1$  to  $\frac{n}{2}$  in parallel do
     $x[2i] = x'[i]$ 
    if  $i \neq \frac{n}{2}$  then
       $x[2i + 1] = x[2i + 1] + x'[i]$ 
```

Segmented scan via prefix sums

Scan

$$y_i = \begin{cases} x_1 & \text{if } i = 1 \\ y_{i-1} \oplus x_i & \text{if } i > 1 \end{cases}$$

Segmented scan

$$\begin{pmatrix} y_i \\ b'_i \end{pmatrix} = \begin{cases} \begin{pmatrix} x_1 \\ b_1 \end{pmatrix} & \text{if } i = 1 \\ \begin{pmatrix} y_{i-1} \\ b'_{i-1} \end{pmatrix} \bullet \begin{pmatrix} x_i \\ b_i \end{pmatrix} & \text{if } i > 1 \end{cases}$$

Claim. • *is associative*

```
procedure SCAN( $x[1..n]$ ,  $\oplus$ )
  if  $n \leq 1$  then return
  for  $i = 1$  to  $\frac{n}{2}$  in parallel do
     $x'[i] = x[2i - 1] \oplus x[2i]$ 
    SCAN( $x'[1..\frac{n}{2}]$ )
  for  $i = 1$  to  $\frac{n}{2}$  in parallel do
     $x[2i] = x'[i]$ 
    if  $i \neq \frac{n}{2}$  then
       $x[2i + 1] = x[2i + 1] \oplus x'[i]$ 
```

Segmented scan via prefix sums

Scan

$$y_i = \begin{cases} x_1 & \text{if } i = 1 \\ y_{i-1} \oplus x_i & \text{if } i > 1 \end{cases}$$

Segmented scan

$$\begin{pmatrix} y_i \\ b'_i \end{pmatrix} = \begin{cases} \begin{pmatrix} x_1 \\ b_1 \end{pmatrix} & \text{if } i = 1 \\ \begin{pmatrix} y_{i-1} \\ b'_{i-1} \end{pmatrix} \bullet \begin{pmatrix} x_i \\ b_i \end{pmatrix} & \text{if } i > 1 \end{cases}$$

Claim. • *is associative*

```
procedure SCAN( $x[1..n]$ ,  $\oplus$ )
if  $n \leq 1$  then return
for  $i = 1$  to  $\frac{n}{2}$  in parallel do
     $x'[i] = x[2i - 1] \oplus x[2i]$ 
    SCAN( $x'[1..\frac{n}{2}]$ )
    for  $i = 1$  to  $\frac{n}{2}$  in parallel do
         $x[2i] = x'[i]$ 
        if  $i \neq \frac{n}{2}$  then
             $x[2i + 1] = x[2i + 1] \oplus x'[i]$ 
```

```
procedure SEG-SCAN( $x[1..n]$ ,  $b[1..n]$ ,  $\oplus$ )
```

Segmented scan via prefix sums

Scan

$$y_i = \begin{cases} x_1 & \text{if } i = 1 \\ y_{i-1} \oplus x_i & \text{if } i > 1 \end{cases}$$

Segmented scan

$$\begin{pmatrix} y_i \\ b'_i \end{pmatrix} = \begin{cases} \begin{pmatrix} x_1 \\ b_1 \end{pmatrix} & \text{if } i = 1 \\ \begin{pmatrix} y_{i-1} \\ b'_{i-1} \end{pmatrix} \bullet \begin{pmatrix} x_i \\ b_i \end{pmatrix} & \text{if } i > 1 \end{cases}$$

class PAIR
double first
bit second

Claim. • *is associative*

```
procedure SCAN( $x[1..n]$ ,  $\oplus$ )
if  $n \leq 1$  then return
for  $i = 1$  to  $\frac{n}{2}$  in parallel do
     $x'[i] = x[2i - 1] \oplus x[2i]$ 
    SCAN( $x'[1..\frac{n}{2}]$ )
    for  $i = 1$  to  $\frac{n}{2}$  in parallel do
         $x[2i] = x'[i]$ 
        if  $i \neq \frac{n}{2}$  then
             $x[2i + 1] = x[2i + 1] \oplus x'[i]$ 
```

```
procedure SEG-SCAN( $x[1..n]$ ,  $b[1..n]$ ,  $\oplus$ )
```

Segmented scan via prefix sums

Scan

$$y_i = \begin{cases} x_1 & \text{if } i = 1 \\ y_{i-1} \oplus x_i & \text{if } i > 1 \end{cases}$$

Segmented scan

$$\begin{pmatrix} y_i \\ b'_i \end{pmatrix} = \begin{cases} \begin{pmatrix} x_1 \\ b_1 \end{pmatrix} & \text{if } i = 1 \\ \begin{pmatrix} y_{i-1} \\ b'_{i-1} \end{pmatrix} \bullet \begin{pmatrix} x_i \\ b_i \end{pmatrix} & \text{if } i > 1 \end{cases}$$

class PAIR
double first
bit second

Claim. • *is associative*

```
procedure SCAN( $x[1..n]$ ,  $\oplus$ )
if  $n \leq 1$  then return
for  $i = 1$  to  $\frac{n}{2}$  in parallel do
     $x'[i] = x[2i - 1] \oplus x[2i]$ 
    SCAN( $x'[1..\frac{n}{2}]$ )
    for  $i = 1$  to  $\frac{n}{2}$  in parallel do
         $x[2i] = x'[i]$ 
        if  $i \neq \frac{n}{2}$  then
             $x[2i + 1] = x[2i + 1] \oplus x'[i]$ 
```

```
procedure SEG-SCAN( $x[1..n]$ ,  $b[1..n]$ ,  $\oplus$ )
 $X[1..n]$  = new array of PAIRS
for  $i = 1$  to  $n$  in parallel do
     $X[i].first = x[i]$ 
     $X[i].second = b[i]$ 
```

Segmented scan via prefix sums

Scan

$$y_i = \begin{cases} x_1 & \text{if } i = 1 \\ y_{i-1} \oplus x_i & \text{if } i > 1 \end{cases}$$

Segmented scan

$$\begin{pmatrix} y_i \\ b'_i \end{pmatrix} = \begin{cases} \begin{pmatrix} x_1 \\ b_1 \end{pmatrix} & \text{if } i = 1 \\ \begin{pmatrix} y_{i-1} \\ b'_{i-1} \end{pmatrix} \bullet \begin{pmatrix} x_i \\ b_i \end{pmatrix} & \text{if } i > 1 \end{cases}$$

```
class PAIR  
double first  
bit second
```

Claim. • *is associative*

```
procedure SCAN(x[1..n],  $\oplus$ )  
if  $n \leq 1$  then return  
for  $i = 1$  to  $\frac{n}{2}$  in parallel do  
     $x'[i] = x[2i - 1] \oplus x[2i]$   
    SCAN( $x'[1..\frac{n}{2}]$ )  
    for  $i = 1$  to  $\frac{n}{2}$  in parallel do  
         $x[2i] = x'[i]$   
        if  $i \neq \frac{n}{2}$  then  
             $x[2i + 1] = x[2i + 1] \oplus x'[i]$ 
```

```
procedure SEG-SCAN(x[1..n], b[1..n],  $\oplus$ )  
 $X[1..n]$  = new array of PAIRS  
for  $i = 1$  to  $n$  in parallel do  
     $X[i].first = x[i]$   
     $X[i].second = b[i]$   
operator  $\bullet$ (PAIR, PAIR,  $\oplus$ )
```

Segmented scan via prefix sums

Scan

$$y_i = \begin{cases} x_1 & \text{if } i = 1 \\ y_{i-1} \oplus x_i & \text{if } i > 1 \end{cases}$$

Segmented scan

$$\begin{pmatrix} y_i \\ b'_i \end{pmatrix} = \begin{cases} \begin{pmatrix} x_1 \\ b_1 \end{pmatrix} & \text{if } i = 1 \\ \begin{pmatrix} y_{i-1} \\ b'_{i-1} \end{pmatrix} \bullet \begin{pmatrix} x_i \\ b_i \end{pmatrix} & \text{if } i > 1 \end{cases}$$

```
class PAIR  
double first  
bit second
```

Claim. • *is associative*

```
procedure SCAN(x[1..n],  $\oplus$ )  
if  $n \leq 1$  then return  
for  $i = 1$  to  $\frac{n}{2}$  in parallel do  
     $x'[i] = x[2i - 1] \oplus x[2i]$   
    SCAN( $x'[1..\frac{n}{2}]$ )  
    for  $i = 1$  to  $\frac{n}{2}$  in parallel do  
         $x[2i] = x'[i]$   
        if  $i \neq \frac{n}{2}$  then  
             $x[2i + 1] = x[2i + 1] \oplus x'[i]$ 
```

```
procedure SEG-SCAN(x[1..n], b[1..n],  $\oplus$ )  
 $X[1..n]$  = new array of PAIRS  
for  $i = 1$  to  $n$  in parallel do  
     $X[i].first = x[i]$   
     $X[i].second = b[i]$   
operator  $\bullet$ (PAIR, PAIR,  $\oplus$ )  
SCAN( $X[1..n]$ ,  $\bullet$ )
```

Segmented scan via prefix sums

Scan

$$y_i = \begin{cases} x_1 & \text{if } i = 1 \\ y_{i-1} \oplus x_i & \text{if } i > 1 \end{cases}$$

Segmented scan

$$\begin{pmatrix} y_i \\ b'_i \end{pmatrix} = \begin{cases} \begin{pmatrix} x_1 \\ b_1 \end{pmatrix} & \text{if } i = 1 \\ \begin{pmatrix} y_{i-1} \\ b'_{i-1} \end{pmatrix} \bullet \begin{pmatrix} x_i \\ b_i \end{pmatrix} & \text{if } i > 1 \end{cases}$$

class PAIR
double first
bit second

Claim. • *is associative*

```
procedure SCAN( $x[1..n]$ ,  $\oplus$ )
if  $n \leq 1$  then return
for  $i = 1$  to  $\frac{n}{2}$  in parallel do
     $x'[i] = x[2i - 1] \oplus x[2i]$ 
    SCAN( $x'[1..\frac{n}{2}]$ )
    for  $i = 1$  to  $\frac{n}{2}$  in parallel do
         $x[2i] = x'[i]$ 
        if  $i \neq \frac{n}{2}$  then
             $x[2i + 1] = x[2i + 1] \oplus x'[i]$ 
```

```
procedure SEG-SCAN( $x[1..n]$ ,  $b[1..n]$ ,  $\oplus$ )
 $X[1..n]$  = new array of PAIRS
for  $i = 1$  to  $n$  in parallel do
     $X[i].first = x[i]$ 
     $X[i].second = b[i]$ 
operator  $\bullet$ (PAIR, PAIR,  $\oplus$ )
    SCAN( $X[1..n]$ ,  $\bullet$ )
    for  $i = 1$  to  $n$  in parallel do
         $x[i] = X[i].first$ 
```

Defining •

$$\bullet : (\mathbb{R} \times \{0, 1\}) \times (\mathbb{R} \times \{0, 1\}) \rightarrow (\mathbb{R} \times \{0, 1\})$$

$$\begin{pmatrix} x \\ b_1 \end{pmatrix} \bullet \begin{pmatrix} y \\ b_2 \end{pmatrix} = \begin{pmatrix} (x \otimes b_2) \oplus y \\ b_1 \vee b_2 \end{pmatrix}$$

operator \bullet (PAIR x , PAIR y , **operator** \oplus)

$$\otimes : \mathbb{R} \times \{0, 1\} \rightarrow \mathbb{R}$$

$$x \otimes b = \begin{cases} x & \text{if } b = 0 \\ I_{\oplus} & \text{if } b = 1 \end{cases}$$

Defining •

$$\bullet : (\mathbb{R} \times \{0, 1\}) \times (\mathbb{R} \times \{0, 1\}) \rightarrow (\mathbb{R} \times \{0, 1\})$$

$$\begin{pmatrix} x \\ b_1 \end{pmatrix} \bullet \begin{pmatrix} y \\ b_2 \end{pmatrix} = \begin{pmatrix} (x \otimes b_2) \oplus y \\ b_1 \vee b_2 \end{pmatrix}$$

operator \bullet (PAIR x , PAIR y , **operator** \oplus)

$z = \text{new PAIR}$

$$\otimes : \mathbb{R} \times \{0, 1\} \rightarrow \mathbb{R}$$

$$x \otimes b = \begin{cases} x & \text{if } b = 0 \\ I_{\oplus} & \text{if } b = 1 \end{cases}$$

Defining •

$$\bullet : (\mathbb{R} \times \{0, 1\}) \times (\mathbb{R} \times \{0, 1\}) \rightarrow (\mathbb{R} \times \{0, 1\})$$

$$\begin{pmatrix} x \\ b_1 \end{pmatrix} \bullet \begin{pmatrix} y \\ b_2 \end{pmatrix} = \begin{pmatrix} (x \otimes b_2) \oplus y \\ b_1 \vee b_2 \end{pmatrix}$$

operator \bullet (PAIR x , PAIR y , **operator** \oplus)

$z = \text{new PAIR}$

if $y.\text{second} = 0$ **then**

$z.\text{first} = x.\text{first} \oplus y.\text{first}$

else

$z.\text{first} = I_{\oplus} \oplus y.\text{first}$

$$\otimes : \mathbb{R} \times \{0, 1\} \rightarrow \mathbb{R}$$

$$x \otimes b = \begin{cases} x & \text{if } b = 0 \\ I_{\oplus} & \text{if } b = 1 \end{cases}$$

Defining •

$$\bullet : (\mathbb{R} \times \{0, 1\}) \times (\mathbb{R} \times \{0, 1\}) \rightarrow (\mathbb{R} \times \{0, 1\})$$

$$\begin{pmatrix} x \\ b_1 \end{pmatrix} \bullet \begin{pmatrix} y \\ b_2 \end{pmatrix} = \begin{pmatrix} (x \otimes b_2) \oplus y \\ b_1 \vee b_2 \end{pmatrix}$$

operator \bullet (PAIR x , PAIR y , **operator** \oplus)

$z = \text{new PAIR}$

if $y.\text{second} = 0$ **then**

$z.\text{first} = x.\text{first} \oplus y.\text{first}$

else

$z.\text{first} = I_{\oplus} \oplus y.\text{first}$

$z.\text{second} = x.\text{second} \vee y.\text{second}$

$$\otimes : \mathbb{R} \times \{0, 1\} \rightarrow \mathbb{R}$$

$$x \otimes b = \begin{cases} x & \text{if } b = 0 \\ I_{\oplus} & \text{if } b = 1 \end{cases}$$

Defining •

$$\bullet : (\mathbb{R} \times \{0, 1\}) \times (\mathbb{R} \times \{0, 1\}) \rightarrow (\mathbb{R} \times \{0, 1\})$$

$$\begin{pmatrix} x \\ b_1 \end{pmatrix} \bullet \begin{pmatrix} y \\ b_2 \end{pmatrix} = \begin{pmatrix} (x \otimes b_2) \oplus y \\ b_1 \vee b_2 \end{pmatrix}$$

operator \bullet (PAIR x , PAIR y , **operator** \oplus)

$z = \text{new PAIR}$

if $y.\text{second} = 0$ **then**

$z.\text{first} = x.\text{first} \oplus y.\text{first}$

else

$z.\text{first} = I_{\oplus} \oplus y.\text{first}$

$z.\text{second} = x.\text{second} \vee y.\text{second}$

return z

$$\otimes : \mathbb{R} \times \{0, 1\} \rightarrow \mathbb{R}$$

$$x \otimes b = \begin{cases} x & \text{if } b = 0 \\ I_{\oplus} & \text{if } b = 1 \end{cases}$$

Defining •

$$\bullet : (\mathbb{R} \times \{0, 1\}) \times (\mathbb{R} \times \{0, 1\}) \rightarrow (\mathbb{R} \times \{0, 1\})$$

$$\begin{pmatrix} x \\ b_1 \end{pmatrix} \bullet \begin{pmatrix} y \\ b_2 \end{pmatrix} = \begin{pmatrix} (x \otimes b_2) \oplus y \\ b_1 \vee b_2 \end{pmatrix}$$

operator \bullet (PAIR x , PAIR y , **operator** \oplus)

$z = \text{new PAIR}$

if $y.\text{second} = 0$ **then**

$z.\text{first} = x.\text{first} \oplus y.\text{first}$

else

$z.\text{first} = y.\text{first}$

$z.\text{second} = x.\text{second} \vee y.\text{second}$

return z

$$\otimes : \mathbb{R} \times \{0, 1\} \rightarrow \mathbb{R}$$

$$x \otimes b = \begin{cases} x & \text{if } b = 0 \\ I_{\oplus} & \text{if } b = 1 \end{cases}$$

Defining Segmented Applications

- Define $\oplus : \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$
 - Associative operator
 - Left identity I_{\oplus}

$$(x \oplus y) \oplus z = x \oplus (y \oplus z)$$

$$I_{\oplus} \otimes x = x$$

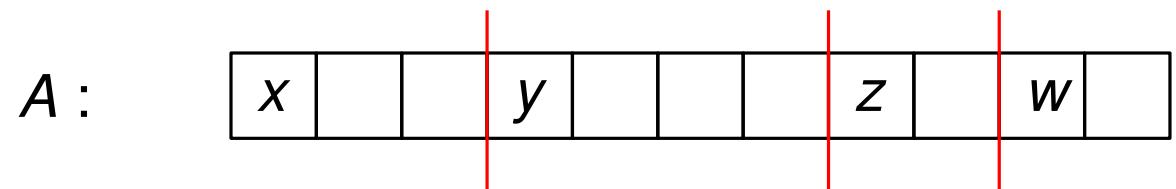
Defining Segmented Applications

- Define $\oplus : \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$
 - Associative operator
 - Left identity I_{\oplus}
- Run SEG-SCAN with operator \oplus

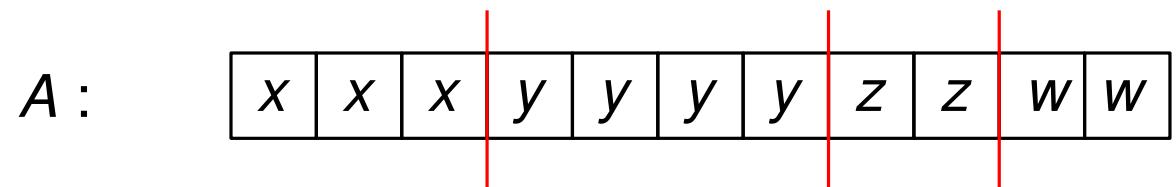
$$(x \oplus y) \oplus z = x \oplus (y \oplus z)$$

$$I_{\oplus} \otimes x = x$$

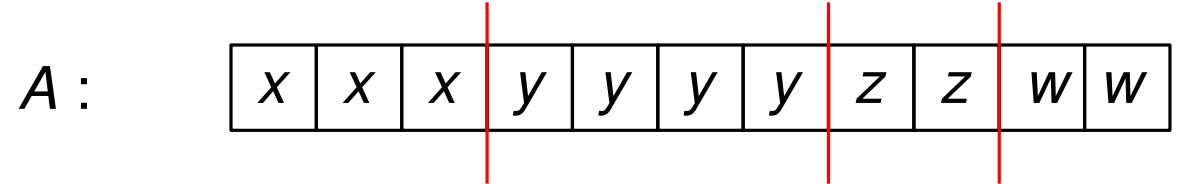
Segmented BROADCAST



Segmented BROADCAST



Segmented BROADCAST



- Define $\oplus : \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$
 - Associative operator
 - Left identity I_{\oplus}

Segmented BROADCAST

```
operator COPY'(x, y)
  if x = ⊥ then
    return y
  else
    return x
```

A :

x	x	x	y	y	y	y	z	z	w	w
---	---	---	---	---	---	---	---	---	---	---

- Define $\oplus : \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$
 - Associative operator
 - Left identity I_{\oplus}

- Define $COPY' : \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$
 - Associative operator
 - Left identity $I_{\oplus} = \perp$

Segmented BROADCAST

```
operator COPY'(x, y)
  if  $x = \perp$  then
    return  $y$ 
  else
    return  $x$ 
```

$A :$

x	x	x	y	y	y	y	z	z	w	w
---	---	---	---	---	---	---	---	---	---	---

- Define $\oplus : \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$
 - Associative operator
 - Left identity I_{\oplus}

- Define $COPY' : \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$
 - Associative operator
 - Left identity $I_{\oplus} = \perp$



$$COPY(a, COPY(b, c)) = COPY(COPY(a, b), c)$$
$$COPY(\perp, x) = x$$

Segmented BROADCAST

```
operator COPY'(x, y)
  if  $x = \perp$  then
    return  $y$ 
  else
    return  $x$ 
```

$A :$

x	x	x	y	y	y	y	z	z	w	w
---	---	---	---	---	---	---	---	---	---	---

- Define $\oplus : \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$
 - Associative operator
 - Left identity I_{\oplus}

- Define $COPY' : \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$
 - Associative operator ✓ $COPY(a, COPY(b, c)) = COPY(COPY(a, b), c)$
 - Left identity $I_{\oplus} = \perp$ ✓ $COPY(\perp, x) = x$

SEG-BROADCAST($A[1..n], b[1..n]$)

SEG-SCAN($A[1..n], b[1..n], \text{COPY}'$)

Segmented BROADCAST

```
operator COPY'(x, y)
  if  $x = \perp$  then
    return  $y$ 
  else
    return  $x$ 
```

$A :$

x	x	x	y	y	y	y	z	z	w	w
---	---	---	---	---	---	---	---	---	---	---

- Define $\oplus : \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$
 - Associative operator
 - Left identity I_{\oplus}

- Define $COPY' : \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$
 - Associative operator
 - Left identity $I_{\oplus} = \perp$

procedure SEG-SCAN($x[1..n]$, $b[1..n]$, \oplus)

$X[1..n]$ = new array of PAIRS

for $i = 1$ to n in parallel do

$X[i].first = x[i]$

$X[i].second = b[i]$

operator \bullet (PAIR, PAIR, \oplus)

SCAN($X[1..n]$, \bullet)

for $i = 1$ to n in parallel do

$x[i] = X[i].first$

SEG-BROADCAST($A[1..n]$, $b[1..n]$)

SEG-SCAN($A[1..n]$, $b[1..n]$, COPY')

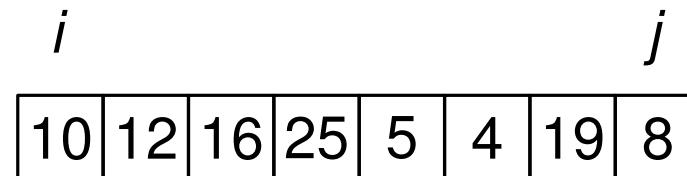
QUICKSORT($A[1..n]$)

QUICKSORT($A[1..n]$)

```
procedure QUICKSORT( $A[i..j]$ )
  if  $i < j$  then
    pivot = RANDOM( $i, j$ )
    SWAP( $A[i], A[pivot]$ )
     $k = \text{PARTITION}(A[i..j])$ 
    in parallel do
      QUICKSORT( $A[i..k - 1]$ )
      QUICKSORT( $A[k + 1..j]$ )
```

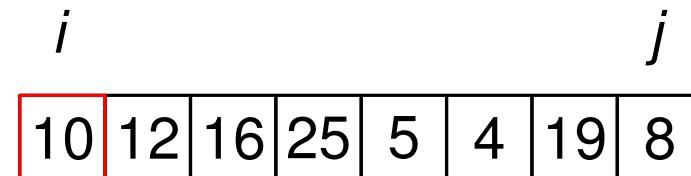
QUICKSORT($A[1..n]$)

```
procedure QUICKSORT( $A[i..j]$ )
  if  $i < j$  then
    pivot = RANDOM( $i, j$ )
    SWAP( $A[i], A[pivot]$ )
     $k = \text{PARTITION}(A[i..j])$ 
    in parallel do
      QUICKSORT( $A[i..k - 1]$ )
      QUICKSORT( $A[k + 1..j]$ )
```



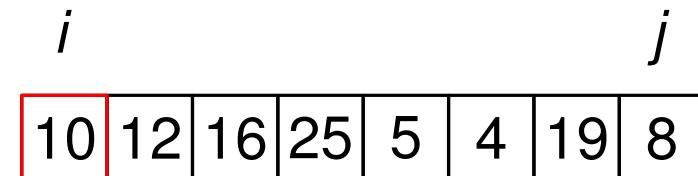
QUICKSORT($A[1..n]$)

```
procedure QUICKSORT( $A[i..j]$ )
  if  $i < j$  then
    pivot = RANDOM( $i, j$ )
    SWAP( $A[i], A[pivot]$ )
     $k = \text{PARTITION}(A[i..j])$ 
    in parallel do
      QUICKSORT( $A[i..k - 1]$ )
      QUICKSORT( $A[k + 1..j]$ )
```



QUICKSORT($A[1..n]$)

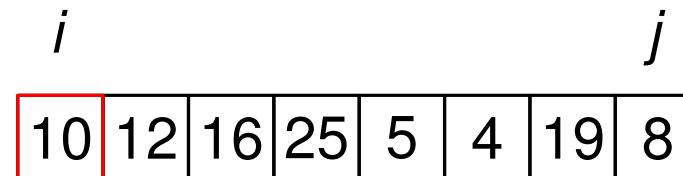
```
procedure QUICKSORT( $A[i..j]$ )
  if  $i < j$  then
    pivot = RANDOM( $i, j$ )
    SWAP( $A[i], A[pivot]$ )
     $k = \text{PARTITION}(A[i..j])$ 
    in parallel do
      QUICKSORT( $A[i..k - 1]$ )
      QUICKSORT( $A[k + 1..j]$ )
```



PARTITION($A[i..j]$)

QUICKSORT($A[1..n]$)

```
procedure QUICKSORT( $A[i..j]$ )
  if  $i < j$  then
    pivot = RANDOM( $i, j$ )
    SWAP( $A[i], A[pivot]$ )
     $k = \text{PARTITION}(A[i..j])$ 
    in parallel do
      QUICKSORT( $A[i..k - 1]$ )
      QUICKSORT( $A[k + 1..j]$ )
```

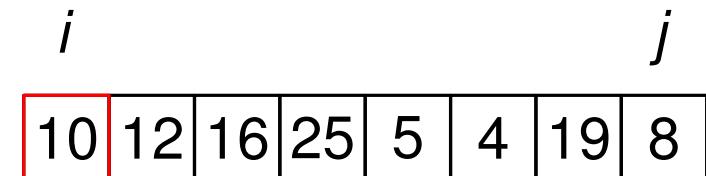


PARTITION($A[i..j]$)

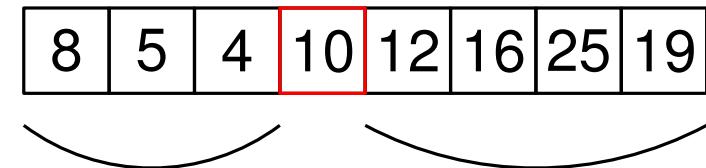


QUICKSORT($A[1..n]$)

```
procedure QUICKSORT( $A[i..j]$ )
  if  $i < j$  then
    pivot = RANDOM( $i, j$ )
    SWAP( $A[i], A[pivot]$ )
     $k = \text{PARTITION}(A[i..j])$ 
    in parallel do
      QUICKSORT( $A[i..k - 1]$ )
      QUICKSORT( $A[k + 1..j]$ )
```



PARTITION($A[i..j]$)

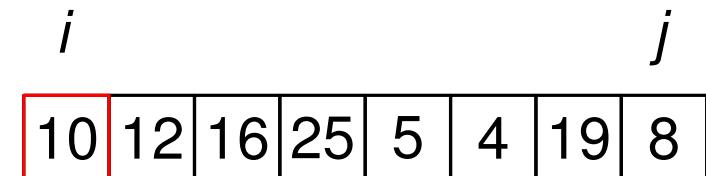


Recurse

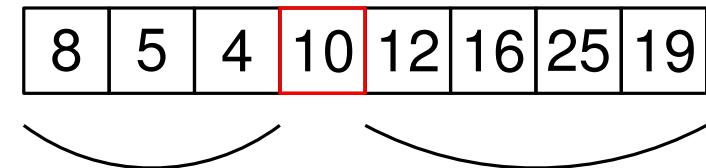
Recurse

QUICKSORT($A[1..n]$)

```
procedure QUICKSORT( $A[i..j]$ )
  if  $i < j$  then
    pivot = RANDOM( $i, j$ )
    SWAP( $A[i], A[pivot]$ )
     $k = \text{PARTITION}(A[i..j])$ 
    in parallel do
      QUICKSORT( $A[i..k - 1]$ )
      QUICKSORT( $A[k + 1..j]$ )
```



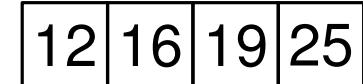
PARTITION($A[i..j]$)



Recurse

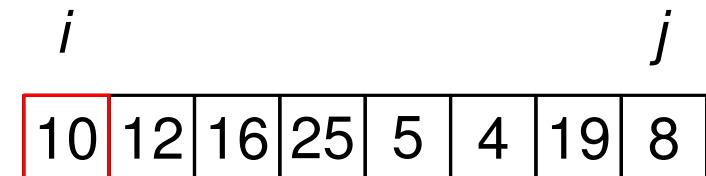


Recurse

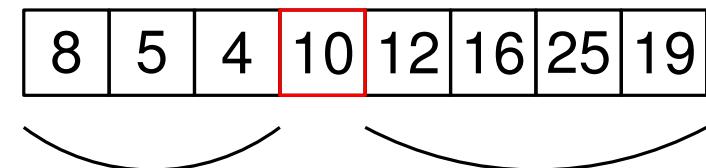


QUICKSORT($A[1..n]$)

```
procedure QUICKSORT( $A[i..j]$ )
  if  $i < j$  then
    pivot = RANDOM( $i, j$ )
    SWAP( $A[i], A[pivot]$ )
     $k = \text{PARTITION}(A[i..j])$ 
    in parallel do
      QUICKSORT( $A[i..k - 1]$ )
      QUICKSORT( $A[k + 1..j]$ )
```



PARTITION($A[i..j]$)

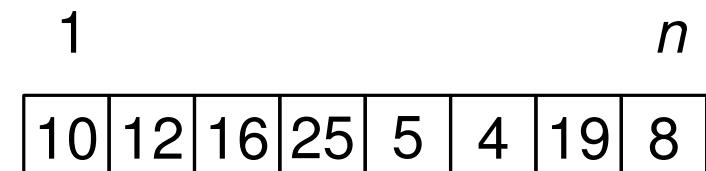


Recurse Recurse



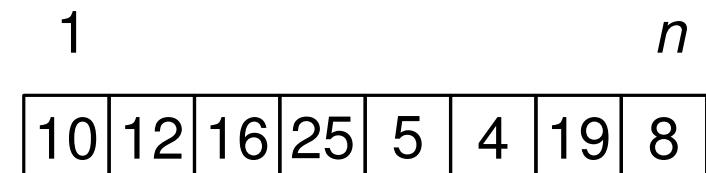
QUICKSORT($A[1..n]$)

```
procedure QUICKSORT( $A[1..n]$ )
    segs = new array of  $n$  bits
    for  $i = 2$  to  $n$  in parallel do segs[ $i$ ] = 0
    segs[1] = 1                                 $\triangleright$  initialize segments
    while not SORTED( $A[1..n]$ ) do
        SEG-RANDOMIZE-PIVOTS( $A$ , segs)
         $p[1..n]$  = SEG-PARTITION( $A[1..n]$ , segs[1.. $n$ ])
        UPDATE-SEGS(segs[1.. $n$ ],  $p[1..n]$ )
```



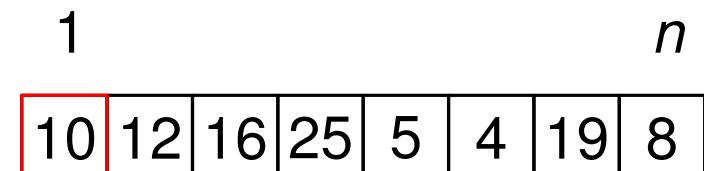
QUICKSORT($A[1..n]$)

```
procedure QUICKSORT( $A[1..n]$ )
    segs = new array of  $n$  bits
    for  $i = 2$  to  $n$  in parallel do  $segs[i] = 0$ 
     $segs[1] = 1$                                 ▷ initialize segments
    while not SORTED( $A[1..n]$ ) do
        SEG-RANDOMIZE-PIVOTS( $A, segs$ )
         $p[1..n] = \text{SEG-PARTITION}(A[1..n], segs[1..n])$ 
        UPDATE-SEGS( $segs[1..n], p[1..n]$ )
```



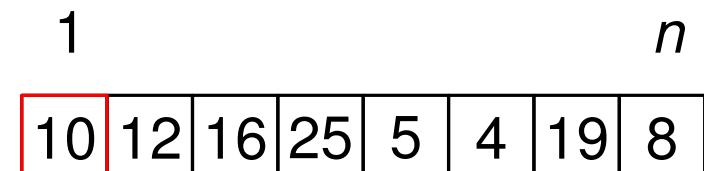
QUICKSORT($A[1..n]$)

```
procedure QUICKSORT( $A[1..n]$ )
    segs = new array of  $n$  bits
    for  $i = 2$  to  $n$  in parallel do  $segs[i] = 0$ 
     $segs[1] = 1$                                 ▷ initialize segments
    while not SORTED( $A[1..n]$ ) do
        SEG-RANDOMIZE-PIVOTS( $A, segs$ )
         $p[1..n] = \text{SEG-PARTITION}(A[1..n], segs[1..n])$ 
        UPDATE-SEGS( $segs[1..n], p[1..n]$ )
```



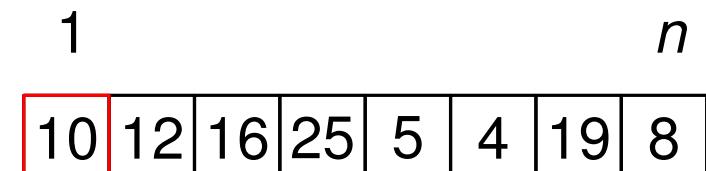
QUICKSORT($A[1..n]$)

```
procedure QUICKSORT( $A[1..n]$ )
     $segs$  = new array of  $n$  bits
    for  $i = 2$  to  $n$  in parallel do  $segs[i] = 0$ 
     $segs[1] = 1$                                 ▷ initialize segments
    while not SORTED( $A[1..n]$ ) do
        SEG-RANDOMIZE-PIVOTS( $A, segs$ )
         $p[1..n] = \text{SEG-PARTITION}(A[1..n], segs[1..n])$ 
        UPDATE-SEGS( $segs[1..n], p[1..n]$ )
```



QUICKSORT($A[1..n]$)

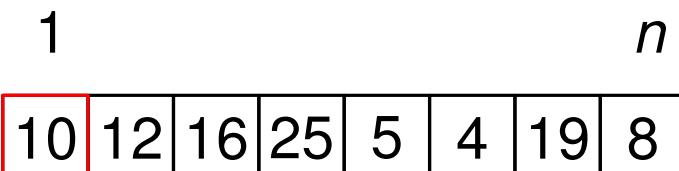
```
procedure QUICKSORT( $A[1..n]$ )
     $segs$  = new array of  $n$  bits
    for  $i = 2$  to  $n$  in parallel do  $segs[i] = 0$ 
     $segs[1] = 1$                                 ▷ initialize segments
    while not SORTED( $A[1..n]$ ) do
        SEG-RANDOMIZE-PIVOTS( $A, segs$ )
         $p[1..n] = \text{SEG-PARTITION}(A[1..n], segs[1..n])$ 
        UPDATE-SEGS( $segs[1..n], p[1..n]$ )
```



SEG-PARTITION($A, segs$)

QUICKSORT($A[1..n]$)

```
procedure QUICKSORT( $A[1..n]$ )
     $segs$  = new array of  $n$  bits
    for  $i = 2$  to  $n$  in parallel do  $segs[i] = 0$ 
     $segs[1] = 1$                                 ▷ initialize segments
    while not SORTED( $A[1..n]$ ) do
        SEG-RANDOMIZE-PIVOTS( $A, segs$ )
         $p[1..n] = \text{SEG-PARTITION}(A[1..n], segs[1..n])$ 
        UPDATE-SEGS( $segs[1..n], p[1..n]$ )
```



SEG-PARTITION($A, segs$)



QUICKSORT($A[1..n]$)

```
procedure QUICKSORT( $A[1..n]$ )
     $segs$  = new array of  $n$  bits
    for  $i = 2$  to  $n$  in parallel do  $segs[i] = 0$ 
     $segs[1] = 1$                                 ▷ initialize segments
    while not SORTED( $A[1..n]$ ) do
        SEG-RANDOMIZE-PIVOTS( $A, segs$ )
         $p[1..n] = \text{SEG-PARTITION}(A[1..n], segs[1..n])$ 
        UPDATE-SEGS( $segs[1..n], p[1..n]$ )
```

1										n
10	12	16	25	5	4	19	8			

SEG-PARTITION($A, segs$)

8	5	4	10	12	16	25	19
0	0	0	1	0	0	0	0

$p : 0 \ 0 \ 0 \ 1 \ 0 \ 0 \ 0 \ 0 \ 0$

QUICKSORT($A[1..n]$)

```
procedure QUICKSORT( $A[1..n]$ )
     $segs$  = new array of  $n$  bits
    for  $i = 2$  to  $n$  in parallel do  $segs[i] = 0$ 
     $segs[1] = 1$                                 ▷ initialize segments
    while not SORTED( $A[1..n]$ ) do
        SEG-RANDOMIZE-PIVOTS( $A, segs$ )
         $p[1..n] = \text{SEG-PARTITION}(A[1..n], segs[1..n])$ 
        UPDATE-SEGS( $segs[1..n], p[1..n]$ )
```

1										n
	10	12	16	25	5	4	19	8		

SEG-PARTITION($A, segs$)

8	5	4	10	12	16	25	19
---	---	---	----	----	----	----	----

$p : 0 \ 0 \ 0 \ 1 \ 0 \ 0 \ 0 \ 0 \ 0$

UPDATE-SEGS($segs, p$)

QUICKSORT($A[1..n]$)

```
procedure QUICKSORT( $A[1..n]$ )
     $segs$  = new array of  $n$  bits
    for  $i = 2$  to  $n$  in parallel do  $segs[i] = 0$ 
     $segs[1] = 1$                                 ▷ initialize segments
    while not SORTED( $A[1..n]$ ) do
        SEG-RANDOMIZE-PIVOTS( $A, segs$ )
         $p[1..n] = \text{SEG-PARTITION}(A[1..n], segs[1..n])$ 
        UPDATE-SEGS( $segs[1..n], p[1..n]$ )
```

1										n
10	12	16	25	5	4	19	8			

SEG-PARTITION($A, segs$)

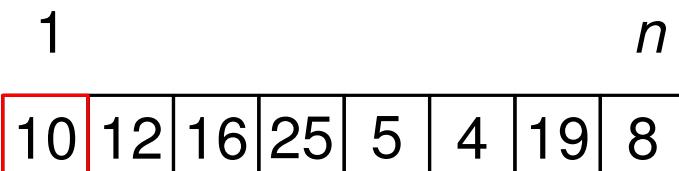
8	5	4	10	12	16	25	19
p : 0	0	0	1	0	0	0	0

UPDATE-SEGS($segs, p$)

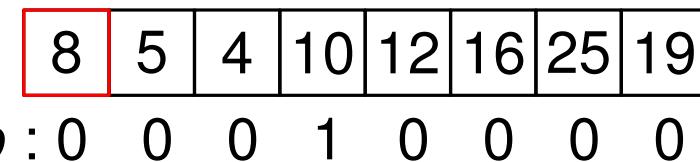
8	5	4	10	12	16	25	19

QUICKSORT($A[1..n]$)

```
procedure QUICKSORT( $A[1..n]$ )
    segs = new array of  $n$  bits
    for  $i = 2$  to  $n$  in parallel do  $segs[i] = 0$ 
     $segs[1] = 1$                                 ▷ initialize segments
    while not SORTED( $A[1..n]$ ) do
        SEG-RANDOMIZE-PIVOTS( $A$ ,  $segs$ )
         $p[1..n] = \text{SEG-PARTITION}(A[1..n], segs[1..n])$ 
        UPDATE-SEGS( $segs[1..n]$ ,  $p[1..n]$ )
```



SEG-PARTITION(A , $segs$)



$p : 0 \ 0 \ 0 \ 1 \ 0 \ 0 \ 0 \ 0 \ 0$

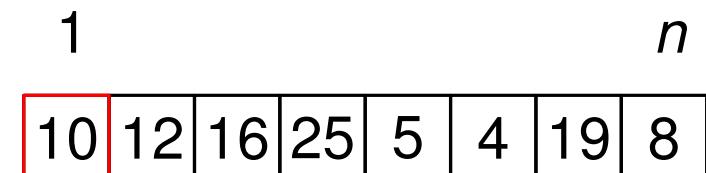
UPDATE-SEGS($segs$, p)



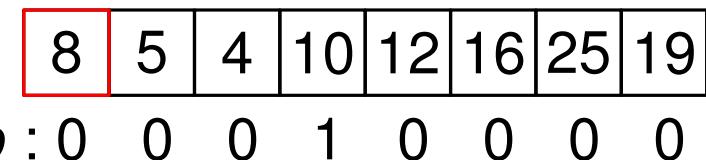
```
procedure UPDATE-SEGS( $segs[1..n]$ ,  $p[1..n]$ )
```

QUICKSORT($A[1..n]$)

```
procedure QUICKSORT( $A[1..n]$ )
    segs = new array of  $n$  bits
    for  $i = 2$  to  $n$  in parallel do  $segs[i] = 0$ 
     $segs[1] = 1$                                 ▷ initialize segments
    while not SORTED( $A[1..n]$ ) do
        SEG-RANDOMIZE-PIVOTS( $A$ ,  $segs$ )
         $p[1..n] = \text{SEG-PARTITION}(A[1..n], segs[1..n])$ 
        UPDATE-SEGS( $segs[1..n]$ ,  $p[1..n]$ )
```



SEG-PARTITION(A , $segs$)



UPDATE-SEGS($segs$, p)



```
procedure UPDATE-SEGS( $segs[1..n]$ ,  $p[1..n]$ )
    for  $i = 1$  to  $n$  in parallel do
        if  $p[i] = 1$  then
             $segs[i] = 1$ 
             $segs[i + 1] = 1$ 
```

QUICKSORT($A[1..n]$)

```
procedure QUICKSORT( $A[1..n]$ )
    segs = new array of  $n$  bits
    for  $i = 2$  to  $n$  in parallel do  $segs[i] = 0$ 
     $segs[1] = 1$                                 ▷ initialize segments
    while not SORTED( $A[1..n]$ ) do
        SEG-RANDOMIZE-PIVOTS( $A$ ,  $segs$ )
         $p[1..n] = \text{SEG-PARTITION}(A[1..n], segs[1..n])$ 
        UPDATE-SEGS( $segs[1..n]$ ,  $p[1..n]$ )
```

1										n
10	12	16	25	5	4	19	8			

SEG-PARTITION(A , $segs$)

8	5	4	10	12	16	25	19
0	0	0	1	0	0	0	0

$p : 0 \ 0 \ 0 \ 1 \ 0 \ 0 \ 0 \ 0 \ 0$

UPDATE-SEGS($segs$, p)

8	5	4	10	12	16	25	19
0	0	0	1	1	1	1	1

```
procedure UPDATE-SEGS( $segs[1..n]$ ,  $p[1..n]$ )
```

```
for  $i = 1$  to  $n$  in parallel do
    if  $p[i] = 1$  then
         $segs[i] = 1$ 
        if  $i + 1 \leq n$  then
             $segs[i + 1] = 1$ 
```

QUICKSORT($A[1..n]$)

```
procedure QUICKSORT( $A[1..n]$ )
    segs = new array of  $n$  bits
    for  $i = 2$  to  $n$  in parallel do  $segs[i] = 0$ 
     $segs[1] = 1$                                 ▷ initialize segments
    while not SORTED( $A[1..n]$ ) do
        SEG-RANDOMIZE-PIVOTS( $A$ ,  $segs$ )
         $p[1..n] = \text{SEG-PARTITION}(A[1..n], segs[1..n])$ 
        UPDATE-SEGS( $segs[1..n]$ ,  $p[1..n]$ )
```

1										n
10	12	16	25	5	4	19	8			

SEG-PARTITION(A , $segs$)

8	5	4	10	12	16	25	19
0	0	0	1	0	0	0	0

$p : 0 \ 0 \ 0 \ 1 \ 0 \ 0 \ 0 \ 0 \ 0$

UPDATE-SEGS($segs$, p)

8	5	4	10	12	16	25	19
0	0	0	1	0	0	0	0

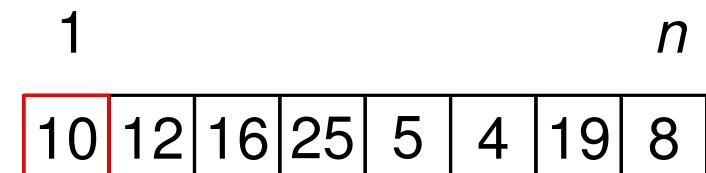
Analysis:

```
procedure UPDATE-SEGS( $segs[1..n]$ ,  $p[1..n]$ )
```

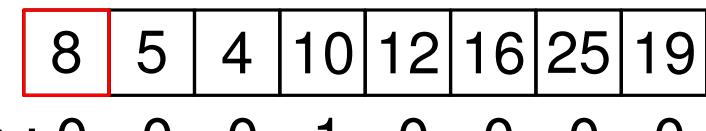
```
for  $i = 1$  to  $n$  in parallel do
    if  $p[i] = 1$  then
         $segs[i] = 1$ 
        if  $i + 1 \leq n$  then
             $segs[i + 1] = 1$ 
```

QUICKSORT($A[1..n]$)

```
procedure QUICKSORT( $A[1..n]$ )
    segs = new array of  $n$  bits
    for  $i = 2$  to  $n$  in parallel do  $segs[i] = 0$ 
     $segs[1] = 1$                                 ▷ initialize segments
    while not SORTED( $A[1..n]$ ) do
        SEG-RANDOMIZE-PIVOTS( $A$ ,  $segs$ )
         $p[1..n] = \text{SEG-PARTITION}(A[1..n], segs[1..n])$ 
        UPDATE-SEGS( $segs[1..n]$ ,  $p[1..n]$ )
```



SEG-PARTITION(A , $segs$)



$p : 0 \ 0 \ 0 \ 1 \ 0 \ 0 \ 0 \ 0 \ 0$

UPDATE-SEGS($segs$, p)



```
procedure UPDATE-SEGS( $segs[1..n]$ ,  $p[1..n]$ )
    for  $i = 1$  to  $n$  in parallel do
        if  $p[i] = 1$  then
             $segs[i] = 1$ 
            if  $i + 1 \leq n$  then
                 $segs[i + 1] = 1$ 
```

Analysis:

$$T(n) = O(1)$$

$$W(n) = O(n)$$

Checking Sortedness

```
procedure SORTED( $A[1..n]$ )
```

5	8	4	10	12	25	16	19
---	---	---	----	----	----	----	----

Checking Sortedness

```
procedure SORTED( $A[1..n]$ )
    sorted = new array of  $n - 1$  booleans
    for  $i = 1$  to  $n - 1$  in parallel do
        if  $A[i] > A[i + 1]$  then
            sorted[ $i$ ] = false
        else
            sorted[ $i$ ] = true
    SCAN(sorted[ $1..n - 1$ ], AND)
    return sorted[ $n - 1$ ]
```

5	8	4	10	12	25	16	19
---	---	---	----	----	----	----	----

Checking Sortedness

```
procedure SORTED( $A[1..n]$ )
     $sorted$  = new array of  $n - 1$  booleans
    for  $i = 1$  to  $n - 1$  in parallel do
        if  $A[i] > A[i + 1]$  then
             $sorted[i] = false$ 
        else
             $sorted[i] = true$ 
    SCAN( $sorted[1..n - 1]$ , AND)
    return  $sorted[n - 1]$ 
```

5	8	4	10	12	25	16	19
---	---	---	----	----	----	----	----

Checking Sortedness

```
procedure SORTED( $A[1..n]$ )
     $sorted$  = new array of  $n - 1$  booleans
    for  $i = 1$  to  $n - 1$  in parallel do
        if  $A[i] > A[i + 1]$  then
             $sorted[i] = false$ 
        else
             $sorted[i] = true$ 
    SCAN( $sorted[1..n - 1]$ , AND)
    return  $sorted[n - 1]$ 
```

5	8	4	10	12	25	16	19
---	---	---	----	----	----	----	----

1

Checking Sortedness

```
procedure SORTED( $A[1..n]$ )
     $sorted$  = new array of  $n - 1$  booleans
    for  $i = 1$  to  $n - 1$  in parallel do
        if  $A[i] > A[i + 1]$  then
             $sorted[i] = false$ 
        else
             $sorted[i] = true$ 
    SCAN( $sorted[1..n - 1]$ , AND)
    return  $sorted[n - 1]$ 
```

5	8	4	10	12	25	16	19
1	0						

Checking Sortedness

```
procedure SORTED( $A[1..n]$ )
    sorted = new array of  $n - 1$  booleans
    for  $i = 1$  to  $n - 1$  in parallel do
        if  $A[i] > A[i + 1]$  then
            sorted[ $i$ ] = false
        else
            sorted[ $i$ ] = true
    SCAN(sorted[1.. $n - 1$ ], AND)
    return sorted[ $n - 1$ ]
```

5	8	4	10	12	25	16	19
1	0	1					

Checking Sortedness

```
procedure SORTED( $A[1..n]$ )
     $sorted$  = new array of  $n - 1$  booleans
    for  $i = 1$  to  $n - 1$  in parallel do
        if  $A[i] > A[i + 1]$  then
             $sorted[i] = false$ 
        else
             $sorted[i] = true$ 
    SCAN( $sorted[1..n - 1]$ , AND)
    return  $sorted[n - 1]$ 
```

5	8	4	10	12	25	16	19
1	0	1	1				

Checking Sortedness

```
procedure SORTED( $A[1..n]$ )
    sorted = new array of  $n - 1$  booleans
    for  $i = 1$  to  $n - 1$  in parallel do
        if  $A[i] > A[i + 1]$  then
            sorted[ $i$ ] = false
        else
            sorted[ $i$ ] = true
    SCAN(sorted[1.. $n - 1$ ], AND)
    return sorted[ $n - 1$ ]
```

5	8	4	10	12	25	16	19
1	0	1	1	1	0	1	

Checking Sortedness

```
procedure SORTED( $A[1..n]$ )
    sorted = new array of  $n - 1$  booleans
    for  $i = 1$  to  $n - 1$  in parallel do
        if  $A[i] > A[i + 1]$  then
            sorted[ $i$ ] = false
        else
            sorted[ $i$ ] = true
    SCAN(sorted[ $1..n - 1$ ], AND)
    return sorted[ $n - 1$ ]
```

5	8	4	10	12	25	16	19
1	0	1	1	1	0	1	

Checking Sortedness

```
procedure SORTED( $A[1..n]$ )
    sorted = new array of  $n - 1$  booleans
    for  $i = 1$  to  $n - 1$  in parallel do
        if  $A[i] > A[i + 1]$  then
            sorted[ $i$ ] = false
        else
            sorted[ $i$ ] = true
    SCAN(sorted[ $1..n - 1$ ], AND)
    return sorted[ $n - 1$ ]
```

5	8	4	10	12	25	16	19
1	0	1	1	1	0	1	
1	0	0	0	0	0	0	0

Checking Sortedness

```
procedure SORTED( $A[1..n]$ )
    sorted = new array of  $n - 1$  booleans
    for  $i = 1$  to  $n - 1$  in parallel do
        if  $A[i] > A[i + 1]$  then
            sorted[ $i$ ] = false
        else
            sorted[ $i$ ] = true
    SCAN(sorted[ $1..n - 1$ ], AND)
    return sorted[ $n - 1$ ]
```

5	8	4	10	12	25	16	19
1	0	1	1	1	0	1	
1	0	0	0	0	0	0	0



Checking Sortedness

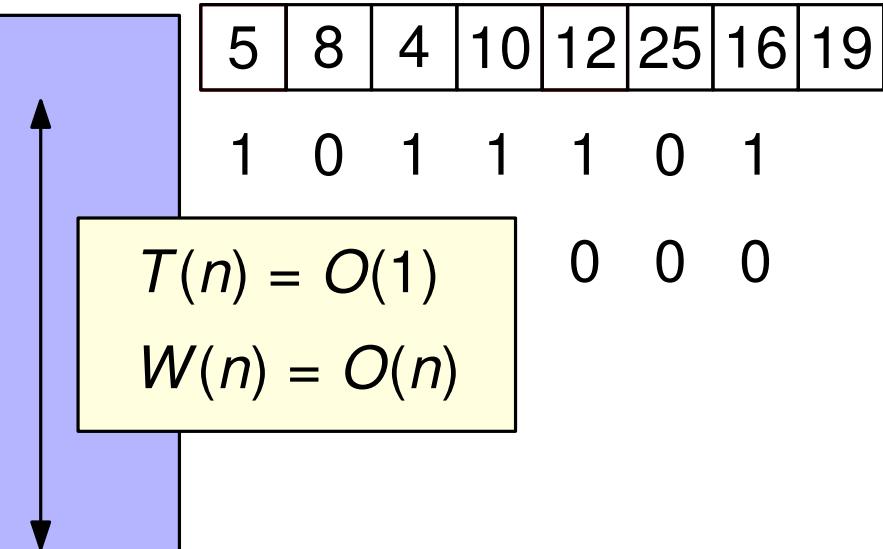
```
procedure SORTED( $A[1..n]$ )
    sorted = new array of  $n - 1$  booleans
    for  $i = 1$  to  $n - 1$  in parallel do
        if  $A[i] > A[i + 1]$  then
            sorted[ $i$ ] = false
        else
            sorted[ $i$ ] = true
    SCAN(sorted[ $1..n - 1$ ], AND)
    return sorted[ $n - 1$ ]
```

5	8	4	10	12	25	16	19
1	0	1	1	1	0	1	
1	0	0	0	0	0	0	0

Analysis:

Checking Sortedness

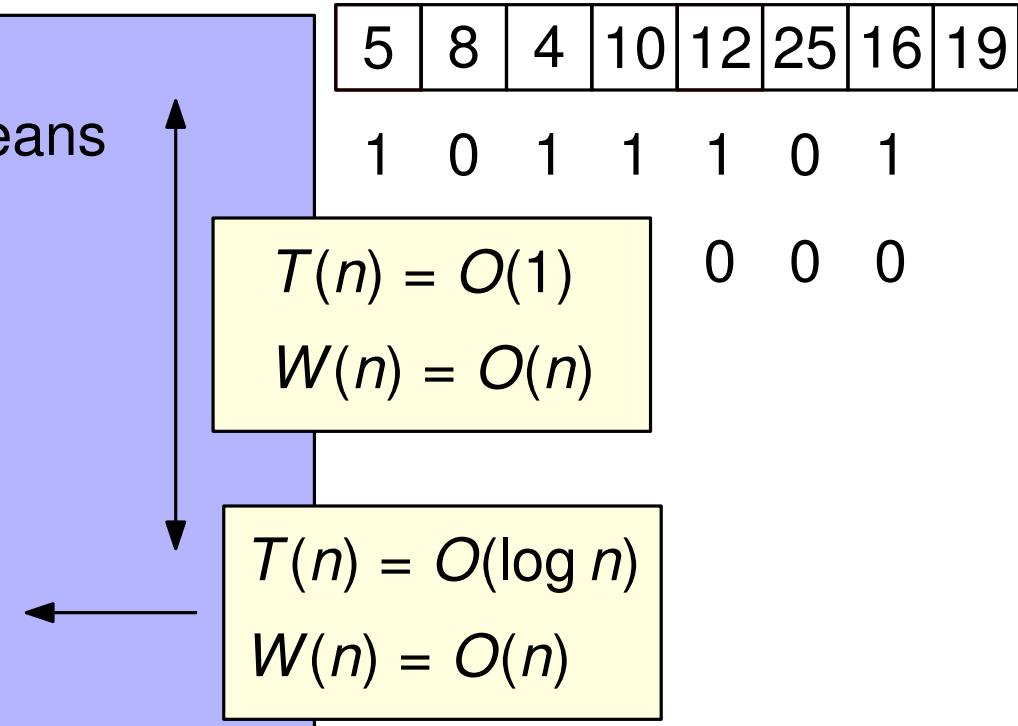
```
procedure SORTED( $A[1..n]$ )
    sorted = new array of  $n - 1$  booleans
    for  $i = 1$  to  $n - 1$  in parallel do
        if  $A[i] > A[i + 1]$  then
            sorted[ $i$ ] = false
        else
            sorted[ $i$ ] = true
    SCAN(sorted[ $1..n - 1$ ], AND)
    return sorted[ $n - 1$ ]
```



Analysis:

Checking Sortedness

```
procedure SORTED( $A[1..n]$ )
    sorted = new array of  $n - 1$  booleans
    for  $i = 1$  to  $n - 1$  in parallel do
        if  $A[i] > A[i + 1]$  then
            sorted[ $i$ ] = false
        else
            sorted[ $i$ ] = true
    SCAN(sorted[1.. $n - 1$ ], AND)
    return sorted[ $n - 1$ ]
```



Analysis:

Checking Sortedness

```
procedure SORTED( $A[1..n]$ )
```

```
    sorted = new array of  $n - 1$  booleans
```

```
    for  $i = 1$  to  $n - 1$  in parallel do
```

```
        if  $A[i] > A[i + 1]$  then
```

```
            sorted[ $i$ ] = false
```

```
        else
```

```
            sorted[ $i$ ] = true
```

```
    SCAN(sorted[1.. $n - 1$ ], AND)
```

```
    return sorted[ $n - 1$ ]
```

5	8	4	10	12	25	16	19
1	0	1	1	1	0	1	

$$T(n) = O(1)$$

$$W(n) = O(n)$$

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

$$W(n) = O(n)$$

Analysis:

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

$$W(n) = O(n)$$

Segmented Partition

```
procedure SEG-PARTITION( $A[1..n]$ ,  $segs[1..n]$ )
```

Segmented Partition

```
procedure SEG-PARTITION( $A[1..n]$ ,  $segs[1..n]$ )
```

	1	2	3	4	5	6	7	8	9	10	11
A	10	12	16	25	5	4	8	7	19	6	18

Segmented Partition

```
procedure SEG-PARTITION( $A[1..n]$ ,  $segs[1..n]$ )
```

	1	2	3	4	5	6	7	8	9	10	11
A	10	12	16	25	5	4	8	7	19	6	18
p	10	10	10	10	10	10	8	8	8	8	8

Segmented Partition

```
procedure SEG-PARTITION( $A[1..n]$ ,  $segs[1..n]$ )
```

$pivots, flags$ = new arrays of size n

for $i = 1$ to n **in parallel do**

if $segs[i] = 1$ **then** $pivots[i] = A[i]$

SEG-BROADCAST($pivots, segs$)

	1	2	3	4	5	6	7	8	9	10	11
A	10	12	16	25	5	4	8	7	19	6	18
p	10						8				

Segmented Partition

```
procedure SEG-PARTITION( $A[1..n]$ ,  $segs[1..n]$ )
```

$pivots, flags$ = new arrays of size n

for $i = 1$ to n **in parallel do**

if $segs[i] = 1$ **then** $pivots[i] = A[i]$

SEG-BROADCAST($pivots, segs$)

	1	2	3	4	5	6	7	8	9	10	11
A	10	12	16	25	5	4	8	7	19	6	18
p	10	10	10	10	10	10	8	8	8	8	8

Segmented Partition

```
procedure SEG-PARTITION( $A[1..n]$ ,  $segs[1..n]$ )
```

$pivots, flags$ = new arrays of size n

for $i = 1$ to n **in parallel do**

if $segs[i] = 1$ **then** $pivots[i] = A[i]$

SEG-BROADCAST($pivots, segs$)

	1	2	3	4	5	6	7	8	9	10	11
A	10	12	16	25	5	4	8	7	19	6	18
p	10	10	10	10	10	10	8	8	8	8	8

Segmented Partition

```
procedure SEG-PARTITION( $A[1..n]$ ,  $segs[1..n]$ )
```

$pivots, flags$ = new arrays of size n

for $i = 1$ to n **in parallel do**

if $segs[i] = 1$ **then** $pivots[i] = A[i]$

SEG-BROADCAST($pivots, segs$)

	1	2	3	4	5	6	7	8	9	10	11
A	10	12	16	25	5	4	8	7	19	6	18
p	10	10	10	10	10	10	8	8	8	8	8
f	1	0	0	0	1	1	1	1	0	1	0

Segmented Partition

```
procedure SEG-PARTITION( $A[1..n]$ ,  $segs[1..n]$ )
```

$pivots, flags$ = new arrays of size n

for $i = 1$ to n **in parallel do**

if $segs[i] = 1$ **then** $pivots[i] = A[i]$

SEG-BROADCAST($pivots, segs$)

for $i = 1$ to n **in parallel do**

if $A[i] \leq pivots[i]$ **then** $flags[i] = 1$

else $flags[i] = 0$

	1	2	3	4	5	6	7	8	9	10	11
A	10	12	16	25	5	4	8	7	19	6	18
p	10	10	10	10	10	10	8	8	8	8	8
f	1	0	0	0	1	1	1	1	0	1	0

Segmented Partition

```
procedure SEG-PARTITION( $A[1..n]$ ,  $segs[1..n]$ )
```

$pivots, flags$ = new arrays of size n

for $i = 1$ to n **in parallel do**

if $segs[i] = 1$ **then** $pivots[i] = A[i]$

SEG-BROADCAST($pivots, segs$)

for $i = 1$ to n **in parallel do**

if $A[i] \leq pivots[i]$ **then** $flags[i] = 1$

else $flags[i] = 0$

	1	2	3	4	5	6	7	8	9	10	11
A	10	12	16	25	5	4	8	7	19	6	18
p	10	10	10	10	10	10	8	8	8	8	8
f	1	0	0	0	1	1	1	1	0	1	0

Segmented Partition

```
procedure SEG-PARTITION( $A[1..n]$ ,  $segs[1..n]$ )
```

$pivots, flags$ = new arrays of size n

for $i = 1$ to n **in parallel do**

if $segs[i] = 1$ **then** $pivots[i] = A[i]$

SEG-BROADCAST($pivots, segs$)

for $i = 1$ to n **in parallel do**

if $A[i] \leq pivots[i]$ **then** $flags[i] = 1$

else $flags[i] = 0$

$k[1..n] = \text{SEG-FILTER}(A[1..n], flags[1..n])$

	1	2	3	4	5	6	7	8	9	10	11
A	10	12	16	25	5	4	8	7	19	6	18
p	10	10	10	10	10	10	8	8	8	8	8
f	1	0	0	0	1	1	1	1	0	1	0

Segmented Partition

```
procedure SEG-PARTITION( $A[1..n]$ ,  $segs[1..n]$ )
```

$pivots, flags$ = new arrays of size n

for $i = 1$ to n **in parallel do**

if $segs[i] = 1$ **then** $pivots[i] = A[i]$

SEG-BROADCAST($pivots, segs$)

for $i = 1$ to n **in parallel do**

if $A[i] \leq pivots[i]$ **then** $flags[i] = 1$

else $flags[i] = 0$

$k[1..n] = \text{SEG-FILTER}(A[1..n], flags[1..n])$

	1	2	3	4	5	6	7	8	9	10	11
A	10	5	4	12	16	25	8	7	6	19	18
p	10	10	10	10	10	10	8	8	8	8	8
f	1	1	1	0	0	0	1	1	1	0	0

Segmented Partition

```
procedure SEG-PARTITION( $A[1..n]$ ,  $segs[1..n]$ )
```

$pivots, flags$ = new arrays of size n

for $i = 1$ to n **in parallel do**

if $segs[i] = 1$ **then** $pivots[i] = A[i]$

SEG-BROADCAST($pivots, segs$)

for $i = 1$ to n **in parallel do**

if $A[i] \leq pivots[i]$ **then** $flags[i] = 1$

else $flags[i] = 0$

$k[1..n] = \text{SEG-FILTER}(A[1..n], flags[1..n])$

Returns: sizes
of new partitions

Details in Homework 3

	1	2	3	4	5	6	7	8	9	10	11
A	10	5	4	12	16	25	8	7	6	19	18
p	10	10	10	10	10	10	8	8	8	8	8
f	1	1	1	0	0	0	1	1	1	0	0

Segmented Partition

```
procedure SEG-PARTITION( $A[1..n]$ ,  $segs[1..n]$ )
```

$pivots, flags$ = new arrays of size n

for $i = 1$ to n **in parallel do**

if $segs[i] = 1$ **then** $pivots[i] = A[i]$

SEG-BROADCAST($pivots, segs$)

for $i = 1$ to n **in parallel do**

if $A[i] \leq pivots[i]$ **then** $flags[i] = 1$

else $flags[i] = 0$

$k[1..n] = \text{SEG-FILTER}(A[1..n], flags[1..n])$

Returns: sizes
of new partitions
Details in Homework 3

	1	2	3	4	5	6	7	8	9	10	11
A	10	5	4	12	16	25	8	7	6	19	18
p	10	10	10	10	10	10	8	8	8	8	8
f	1	1	1	0	0	0	1	1	1	0	0
k	3						3				

Segmented Partition

```
procedure SEG-PARTITION( $A[1..n]$ ,  $segs[1..n]$ )
```

$pivots, flags$ = new arrays of size n

for $i = 1$ to n **in parallel do**

if $segs[i] = 1$ **then** $pivots[i] = A[i]$

SEG-BROADCAST($pivots, segs$)

for $i = 1$ to n **in parallel do**

if $A[i] \leq pivots[i]$ **then** $flags[i] = 1$

else $flags[i] = 0$

$k[1..n] = \text{SEG-FILTER}(A[1..n], flags[1..n])$

Returns: sizes
of new partitions

Details in Homework 3

	1	2	3	4	5	6	7	8	9	10	11
A	10	5	4	12	16	25	8	7	6	19	18
p	10	10	10	10	10	10	8	8	8	8	8
f	1	1	1	0	0	0	1	1	1	0	0
k	3						3				

Segmented Partition

```
procedure SEG-PARTITION( $A[1..n]$ ,  $segs[1..n]$ )
```

$pivots, flags$ = new arrays of size n

for $i = 1$ to n **in parallel do**

if $segs[i] = 1$ **then** $pivots[i] = A[i]$

SEG-BROADCAST($pivots, segs$)

for $i = 1$ to n **in parallel do**

if $A[i] \leq pivots[i]$ **then** $flags[i] = 1$

else $flags[i] = 0$

$k[1..n] = \text{SEG-FILTER}(A[1..n], flags[1..n])$

Returns: sizes
of new partitions
Details in Homework 3

	1	2	3	4	5	6	7	8	9	10	11
A	10	5	4	12	16	25	8	7	6	19	18
p	10	10	10	10	10	10	8	8	8	8	8
f	1	1	1	0	0	0	1	1	1	0	0
k	3						3				

Segmented Partition

```
procedure SEG-PARTITION( $A[1..n]$ ,  $segs[1..n]$ )
```

$pivots, flags$ = new arrays of size n

for $i = 1$ to n **in parallel do**

if $segs[i] = 1$ **then** $pivots[i] = A[i]$

SEG-BROADCAST($pivots, segs$)

for $i = 1$ to n **in parallel do**

if $A[i] \leq pivots[i]$ **then** $flags[i] = 1$

else $flags[i] = 0$

$k[1..n] = \text{SEG-FILTER}(A[1..n], flags[1..n])$

Returns: sizes
of new partitions
Details in Homework 3

	1	2	3	4	5	6	7	8	9	10	11
A	4	5	10	12	16	25	6	7	8	19	18
p	10	10	10	10	10	10	8	8	8	8	8
f	1	1	1	0	0	0	1	1	1	0	0
k	3						3				

Segmented Partition

```
procedure SEG-PARTITION( $A[1..n]$ ,  $segs[1..n]$ )
```

$pivots, flags$ = new arrays of size n

for $i = 1$ to n **in parallel do**

if $segs[i] = 1$ **then** $pivots[i] = A[i]$

SEG-BROADCAST($pivots, segs$)

for $i = 1$ to n **in parallel do**

if $A[i] \leq pivots[i]$ **then** $flags[i] = 1$

else $flags[i] = 0$

$k[1..n] = \text{SEG-FILTER}(A[1..n], flags[1..n])$

for $i = 1$ to n **in parallel do**

if $segs[i] = 1$ **then**

 SWAP($A[i], A[i + k[i] - 1]$)

Returns: sizes
of new partitions
Details in Homework 3

	1	2	3	4	5	6	7	8	9	10	11
A	4	5	10	12	16	25	6	7	8	19	18
p	10	10	10	10	10	10	8	8	8	8	8
f	1	1	1	0	0	0	1	1	1	0	0
k	3						3				

Segmented Partition

```

procedure SEG-PARTITION(  

    pivots, flags = new arrays  

    for i = 1 to n in parallel  

        if segs[i] = 1 then pivots[i] = A[i]  

    SEG-BROADCAST(pivots)  

    for i = 1 to n in parallel  

        if A[i] ≤ pivots[i] then flags[i] = 1  

        else flags[i] = 0

```

k[1..*n*] = SEG-FILTER(*A*[1..*n*], *flags*[1..*n*])

```

for i = 1 to n in parallel do  

    if segs[i] = 1 then  

        SWAP(A[i], A[i + k[i] - 1])

```

```

procedure QUICKSORT(A[1..n])  

    segs = new array of n bits  

    for i = 2 to n in parallel do segs[i] = 0  

    segs[1] = 1                                ▷ initialize segments  

    while not SORTED(A[1..n]) do  

        SEG-RANDOMIZE-PIVOTS(A, segs)  

        p[1..n] = SEG-PARTITION(A[1..n], segs[1..n])  

        UPDATE-SEGS(segs[1..n], p[1..n])

```

PIVOTS

	1	2	3	4	5	6	7	8	9	10	11
<i>A</i>	4	5	10	12	16	25	6	7	8	19	18
<i>p</i>	10	10	10	10	10	10	8	8	8	8	8
<i>f</i>	1	1	1	0	0	0	1	1	1	0	0
<i>k</i>	3						3				

Segmented Partition

```

procedure SEG-PARTITION(  

    pivots, flags = new array of n bits  

    for i = 1 to n in parallel  

        if segs[i] = 1 then pivots[i] = A[i]  

    SEG-BROADCAST(pivots)  

    for i = 1 to n in parallel  

        if A[i]  $\leq$  pivots[i] then flags[i] = 1  

        else flags[i] = 0

```

k[1..*n*] = SEG-FILTER(*A*[1..*n*], *flags*[1..*n*])

```

for i = 1 to n in parallel do  

    if segs[i] = 1 then  

        SWAP(A[i], A[i + k[i] - 1])

```

```

procedure QUICKSORT(A[1..n])  

    segs = new array of n bits  

    for i = 2 to n in parallel do segs[i] = 0  

    segs[1] = 1                                 $\triangleright$  initialize segments  

    while not SORTED(A[1..n]) do  

        SEG-RANDOMIZE-PIVOTS(A, segs)  

        p[1..n] = SEG-PARTITION(A[1..n], segs[1..n])  

        UPDATE-SEGS(segs[1..n], p[1..n])

```

	1	2	3	4	5	6	7	8	9	10	11
<i>A</i>	4	5	10	12	16	25	6	7	8	19	18
<i>p</i>	10	10	10	10	10	10	8	8	8	8	8
<i>f</i>	1	1	1	0	0	0	1	1	1	0	0
<i>k</i>	3						3				
<i>pF</i>	0	0	1	0	0	0	0	0	1	0	0

Segmented Partition

```

procedure SEG-PARTITION(  

    pivots, flags = new arrays of size n  

    for i = 1 to n in parallel  

        if segs[i] = 1 then pivots[i] = A[i]  

    SEG-BROADCAST(pivots)  

    for i = 1 to n in parallel  

        if A[i]  $\leq$  pivots[i] then flags[i] = 1  

        else flags[i] = 0  

k[1..n] = SEG-FILTER(A[1..n], flags[1..n])  

 = new array of size n  

for i = 1 to n in parallel do  

    if segs[i] = 1 then  

        SWAP(A[i], A[i + k[i] - 1])

```

```

procedure QUICKSORT(A[1..n])  

    segs = new array of n bits  

    for i = 2 to n in parallel do segs[i] = 0  

    segs[1] = 1                                 $\triangleright$  initialize segments  

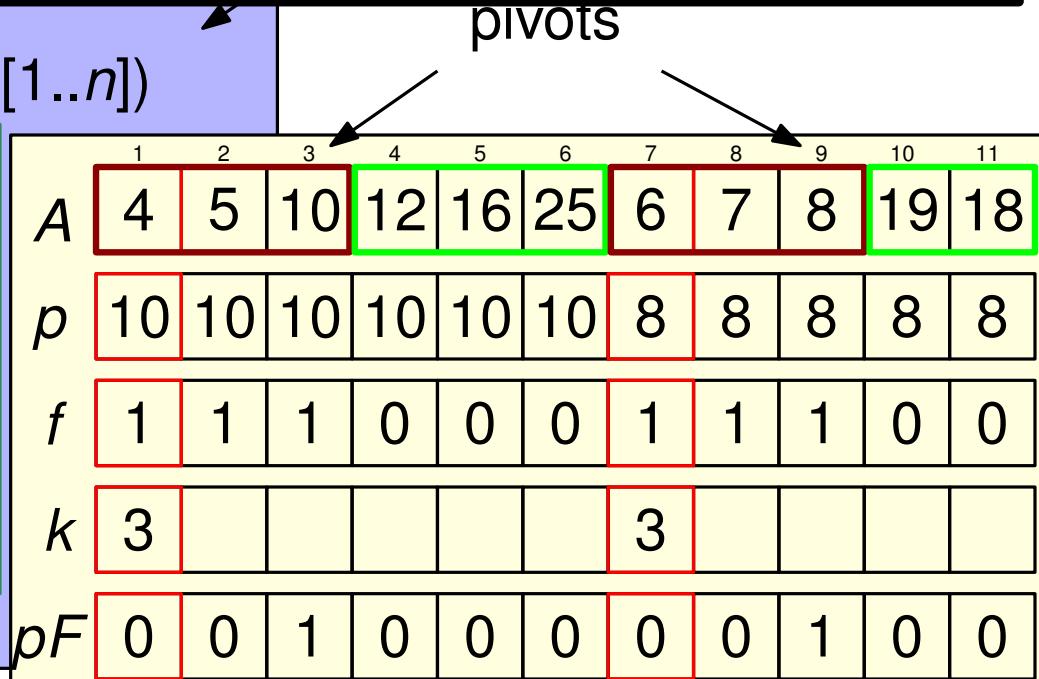
    while not SORTED(A[1..n]) do  

        SEG-RANDOMIZE-PIVOTS(A, segs)  

        p[1..n] = SEG-PARTITION(A[1..n], segs[1..n])  

        UPDATE-SEGS(segs[1..n], p[1..n])

```



The diagram illustrates the state of five arrays over 11 indices (1 to 11). Arrows point from the labels to the corresponding columns in the tables.

	1	2	3	4	5	6	7	8	9	10	11
<i>A</i>	4	5	10	12	16	25	6	7	8	19	18
<i>p</i>	10	10	10	10	10	10	8	8	8	8	8
<i>f</i>	1	1	1	0	0	0	1	1	1	0	0
<i>k</i>	3						3				
<i>pF</i>	0	0	1	0	0	0	0	0	1	0	0

Segmented Partition

```

procedure SEG-PARTITION(  

    pivots, flags = new arrays of size n  

    for i = 1 to n in parallel  

        if segs[i] = 1 then pivots[i] = A[i]  

    SEG-BROADCAST(pivots)  

    for i = 1 to n in parallel  

        if A[i]  $\leq$  pivots[i] then flags[i] = 1  

        else flags[i] = 0  

k[1..n] = SEG-FILTER(A[1..n], flags[1..n])  

 = new array of size n  

for i = 1 to n in parallel do  

    pivotFlags[i] = 0  

    if segs[i] = 1 then  

        SWAP(A[i], A[i + k[i] - 1])  

        pivotFlags[i + k[i] - 1] = 1

```

```

procedure QUICKSORT(A[1..n])  

    segs = new array of n bits  

    for i = 2 to n in parallel do segs[i] = 0  

    segs[1] = 1                                 $\triangleright$  initialize segments  

    while not SORTED(A[1..n]) do  

        SEG-RANDOMIZE-PIVOTS(A, segs)  

        p[1..n] = SEG-PARTITION(A[1..n], segs[1..n])  

        UPDATE-SEGS(segs[1..n], p[1..n])

```

	1	2	3	4	5	6	7	8	9	10	11
<i>A</i>	4	5	10	12	16	25	6	7	8	19	18
<i>p</i>	10	10	10	10	10	10	8	8	8	8	8
<i>f</i>	1	1	1	0	0	0	1	1	1	0	0
<i>k</i>	3						3				
<i>pF</i>	0	0	1	0	0	0	0	0	1	0	0

Segmented Partition

```

procedure SEG-PARTITION(  

    pivots, flags = new arrays of size n  

    for i = 1 to n in parallel  

        if segs[i] = 1 then pivots[i] = A[i]  

    SEG-BROADCAST(pivots)  

    for i = 1 to n in parallel  

        if A[i] ≤ pivots[i] then flags[i] = 1  

        else flags[i] = 0  

k[1..n] = SEG-FILTER(A[1..n], flags[1..n])  

 = new array of size n  

for i = 1 to n in parallel do  

    pivotFlags[i] = 0  

    if segs[i] = 1 then  

        SWAP(A[i], A[i + k[i] - 1])  

        pivotFlags[i + k[i] - 1] = 1  

return pivotFlags[1..n]

```

procedure QUICKSORT(*A*[1..*n*])

segs = new array of *n* bits

for *i* = 2 to *n* **in parallel do** *segs*[*i*] = 0

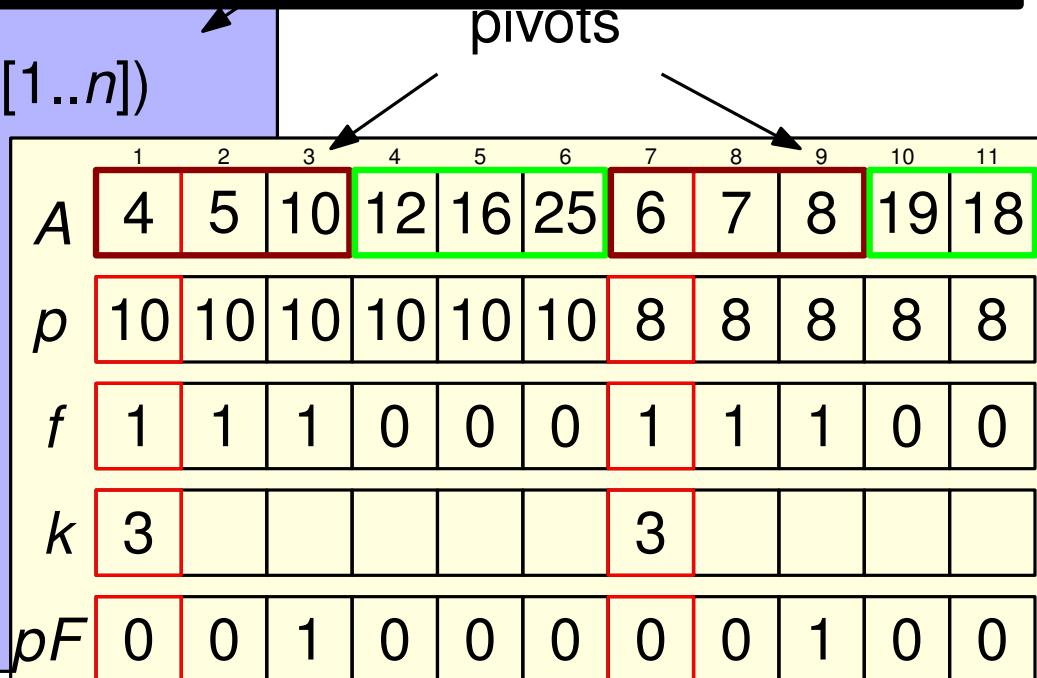
segs[1] = 1 ▷ initialize segments

while not SORTED(*A*[1..*n*]) **do**

 SEG-RANDOMIZE-PIVOTS(*A*, *segs*)

p[1..*n*] = SEG-PARTITION(*A*[1..*n*], *segs*[1..*n*])

 UPDATE-SEGS(*segs*[1..*n*], *p*[1..*n*])



The diagram illustrates the state of five arrays after the first partition step. The arrays are indexed from 1 to 11.

- Array A:** Contains values 4, 5, 10, 12, 16, 25, 6, 7, 8, 19, 18. Cells 4, 5, 10, 12, 16, 25 are highlighted with red borders, while 19 and 18 are highlighted with green borders.
- Array p:** Contains all values 10.
- Array f:** Contains values 1, 1, 1, 0, 0, 0, 1, 1, 1, 0, 0.
- Array k:** Contains values 3, empty, empty, empty, empty, empty, 3, empty, empty, empty.
- Array pF:** Contains values 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0.

Arrows point from the labels *pivots* and *k[1..n]* to the corresponding rows in the arrays. The label *pivots* points to the row where the value 10 is present in array p. The label *k[1..n]* points to the row where the value 3 is present in array k.

Segmented Partition

Analysis:

```
procedure SEG-PARTITION( $A[1..n]$ ,  $segs[1..n]$ )
```

$pivots, flags$ = new arrays of size n

for $i = 1$ to n **in parallel do**

if $segs[i] = 1$ **then** $pivots[i] = A[i]$

SEG-BROADCAST($pivots, segs$)

for $i = 1$ to n **in parallel do**

if $A[i] \leq pivots[i]$ **then** $flags[i] = 1$

else $flags[i] = 0$

$k[1..n] = \text{SEG-FILTER}(A[1..n], flags[1..n])$

$pivotFlags$ = new array of size n

for $i = 1$ to n **in parallel do**

$pivotFlags[i] = 0$

if $segs[i] = 1$ **then**

 SWAP($A[i], A[i + k[i] - 1]$)

$pivotFlags[i + k[i] - 1] = 1$

return $pivotFlags[1..n]$

Segmented Partition

```
procedure SEG-PARTITION( $A[1..n]$ ,  $segs[1..n]$ )
```

$pivots, flags$ = new arrays of size n

for $i = 1$ to n **in parallel do**

if $segs[i] = 1$ **then** $pivots[i] = A[i]$

SEG-BROADCAST($pivots, segs$)

for $i = 1$ to n **in parallel do**

if $A[i] \leq pivots[i]$ **then** $flags[i] = 1$

else $flags[i] = 0$

$k[1..n] = \text{SEG-FILTER}(A[1..n], flags[1..n])$

$pivotFlags$ = new array of size n

for $i = 1$ to n **in parallel do**

$pivotFlags[i] = 0$

if $segs[i] = 1$ **then**

 SWAP($A[i], A[i + k[i] - 1]$)

$pivotFlags[i + k[i] - 1] = 1$

return $pivotFlags[1..n]$

Analysis:

$$T(n) = O(1)$$

$$W(n) = O(n)$$

$$T(n) = O(1)$$

$$W(n) = O(n)$$

$$T(n) = O(1)$$

$$W(n) = O(n)$$

Segmented Partition

Analysis:

```
procedure SEG-PARTITION( $A[1..n]$ ,  $segs[1..n]$ )
```

$pivots, flags$ = new arrays of size n

for $i = 1$ to n **in parallel do**

if $segs[i] = 1$ **then** $pivots[i] = A[i]$

SEG-BROADCAST($pivots, segs$)

for $i = 1$ to n **in parallel do**

if $A[i] \leq pivots[i]$ **then** $flags[i] = 1$

else $flags[i] = 0$

$k[1..n] = \text{SEG-FILTER}(A[1..n], flags[1..n])$

$pivotFlags$ = new array of size n

for $i = 1$ to n **in parallel do**

$pivotFlags[i] = 0$

if $segs[i] = 1$ **then**

 SWAP($A[i], A[i + k[i] - 1]$)

$pivotFlags[i + k[i] - 1] = 1$

return $pivotFlags[1..n]$

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

$$W(n) = O(n)$$

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

$$W(n) = O(n)$$

Segmented Partition

Analysis:

```
procedure SEG-PARTITION( $A[1..n]$ ,  $segs[1..n]$ )
```

$pivots, flags$ = new arrays of size n

for $i = 1$ to n **in parallel do**

if $segs[i] = 1$ **then** $pivots[i] = A[i]$

SEG-BROADCAST($pivots, segs$)

for $i = 1$ to n **in parallel do**

if $A[i] \leq pivots[i]$ **then** $flags[i] = 1$

else $flags[i] = 0$

$k[1..n] = \text{SEG-FILTER}(A[1..n], flags[1..n])$

$pivotFlags$ = new array of size n

for $i = 1$ to n **in parallel do**

$pivotFlags[i] = 0$

if $segs[i] = 1$ **then**

 SWAP($A[i], A[i + k[i] - 1]$)

$pivotFlags[i + k[i] - 1] = 1$

return $pivotFlags[1..n]$

Total:

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

$$W(n) = O(n)$$

Segmented Partition

```
procedure SEG-PARTITION( $A[1..n]$ ,  $segs[1..n]$ )
```

pivots, flags = new arrays of size n

for $i = 1$ to n **in parallel do**

if $segs[i] = 1$

 SEG-BROADCAST($A[i]$)

for $j = 1$ to n **in parallel do**

if $A[j] \leq A[i]$

else $flags[j] = 1$

$k[1..n] = SEG(A)$

$pivotFlags = 0$

for $i = 1$ to n **in parallel do**

$pivotFlags[i] = 0$

if $segs[i] = 1$ **then**

 SWAP($A[i], A[i + k[i] - 1]$)

$pivotFlags[i + k[i] - 1] = 1$

return $pivotFlags[1..n]$

```
procedure QUICKSORT( $A[1..n]$ )
```

$segs$ = new array of n bits

for $i = 2$ to n **in parallel do** $segs[i] = 0$

$segs[1] = 1$ ▷ initialize segments

while not SORTED($A[1..n]$) **do**

 SEG-RANDOMIZE-PIVOTS($A, segs$)

$p[1..n] = SEG-PARTITION(A[1..n], segs[1..n])$

 UPDATE-SEGS($segs[1..n], p[1..n]$)

Segmented Partition

```
procedure SEG-PARTITION( $A[1..n]$ ,  $segs[1..n]$ )
```

pivots, flags = new arrays of size n

for $i = 1$ to n **in parallel do**

if $segs[i] = 1$

 SEG-BROADCAST($A[i]$)

for $i = 1$ to n **in parallel do**

if $A[i] \leq pivot$

else $flags[i] = 1$

$k[1..n] = SEG-AVG(A)$

$pivotFlags = 0$

for $i = 1$ to n **in parallel do**

$pivotFlags[i] = 0$

if $segs[i] = 1$ **then**

 SWAP($A[i], A[i + k[i] - 1]$)

$pivotFlags[i + k[i] - 1] = 1$

return $pivotFlags[1..n]$

```
procedure QUICKSORT( $A[1..n]$ )
```

$segs$ = new array of n bits

for $i = 2$ to n **in parallel do** $segs[i] = 0$

$segs[1] = 1$ ▷ initialize segments

while not SORTED($A[1..n]$) **do**

 SEG-RANDOMIZE-PIVOTS($A, segs$)

$p[1..n] = SEG-PARTITION(A[1..n], segs[1..n])$

 UPDATE-SEGS($segs[1..n], p[1..n]$)

$pivotFlags[1..n] = 0$

if $segs[i] = 1$ **then**

 SWAP($A[i], A[i + k[i] - 1]$)

$pivotFlags[i + k[i] - 1] = 1$

return $pivotFlags[1..n]$

Segmented Partition

```
procedure SEG-PARTITION( $A[1..n]$ ,  $segs[1..n]$ )
```

pivots, flags = new arrays of size n

for $i = 1$ to n **in parallel do**

if $segs[i] = 1$

 SEG-BROADCAST($A[i]$)

for $i = 1$ to n **in parallel do**

if $A[i] \leq pivot$

else $flags[i] = 1$

$k[1..n] = SEG-AVG(A)$

$pivotFlags = 0$

for $i = 1$ to n **in parallel do**

$pivotFlags[i] = 0$

if $segs[i] = 1$ **then**

 SWAP($A[i], A[i + k[i] - 1]$)

$pivotFlags[i + k[i] - 1] = 1$

return $pivotFlags[1..n]$

```
procedure QUICKSORT( $A[1..n]$ )
```

$segs$ = new array of n bits

for $i = 2$ to n **in parallel do** $segs[i] = 0$

$segs[1] = 1$ ▷ initialize segments

while not SORTED($A[1..n]$) **do**

 SEG-RANDOMIZE-PIVOTS($A, segs$)

$p[1..n] = SEG-PARTITION(A[1..n], segs[1..n])$

 UPDATE-SEGS($segs[1..n], p[1..n]$)



Segmented Randomize Pivots

```
procedure SEG-RANDOMIZE-PIVOTS( $A[1..n]$ ,  $segs[1..n]$ )
```

	1	2	3	4	5	6	7	8	9	10	11
A	10	12	16	25	5	4	8	7	19	6	18

Segmented Randomize Pivots

```
procedure SEG-RANDOMIZE-PIVOTS( $A[1..n]$ ,  $segs[1..n]$ )
```

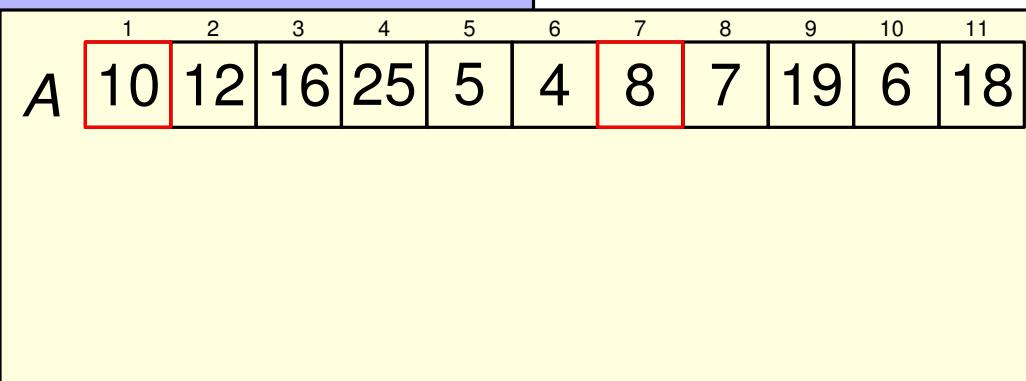
```
for  $i = 1$  to  $n$  in parallel do  
  if  $seg[i] = 1$  then  
     $rnd = RANDOM(start[i], end[i])$   
    SWAP( $A[i]$ ,  $A[rnd]$ )
```

A	1	2	3	4	5	6	7	8	9	10	11
	10	12	16	25	5	4	8	7	19	6	18

Segmented Randomize Pivots

```
procedure SEG-RANDOMIZE-PIVOTS( $A[1..n]$ ,  $segs[1..n]$ )  
     $start, end$  = new arrays of size  $n$ 
```

```
for  $i = 1$  to  $n$  in parallel do  
    if  $seg[i] = 1$  then  
         $rnd = RANDOM(start[i], end[i])$   
        SWAP( $A[i], A[rnd]$ )
```



Segmented Randomize Pivots

```
procedure SEG-RANDOMIZE-PIVOTS( $A[1..n]$ ,  $\text{segs}[1..n]$ )  
     $start, end$  = new arrays of size  $n$ 
```

```
for  $i = 1$  to  $n$  in parallel do  
    if  $\text{seg}[i] = 1$  then  
         $rnd = \text{RANDOM}(start[i], end[i])$   
        SWAP( $A[i], A[rnd]$ )
```

	1	2	3	4	5	6	7	8	9	10	11
A	10	12	16	25	5	4	8	7	19	6	18
s	1						7				
e	6						11				

Segmented Randomize Pivots

```
procedure SEG-RANDOMIZE-PIVOTS( $A[1..n]$ ,  $\text{segs}[1..n]$ )
```

start, end = new arrays of size n

```
for  $i = 1$  to  $n$  in parallel do
```

```
    if  $\text{segs}[i] = 1$  then  $\text{start}[i] = i$ 
```

```
for  $i = 1$  to  $n$  in parallel do
```

```
    if  $\text{seg}[i] = 1$  then
```

```
         $\text{rnd} = \text{RANDOM}(\text{start}[i], \text{end}[i])$ 
```

```
        SWAP( $A[i], A[\text{rnd}]$ )
```

	1	2	3	4	5	6	7	8	9	10	11
A	10	12	16	25	5	4	8	7	19	6	18
s	1						7				
e	6						11				

Segmented Randomize Pivots

```
procedure SEG-RANDOMIZE-PIVOTS( $A[1..n]$ ,  $\text{segs}[1..n]$ )
```

$start, end$ = new arrays of size n

for $i = 1$ to n **in parallel do**

if $\text{segs}[i] = 1$ **then** $start[i] = i$

```
        SEG-BROADCAST( $start[1..n]$ ,  $\text{segs}[1..n]$ )
```

for $i = 1$ to n **in parallel do**

if $\text{seg}[i] = 1$ **then**

$rnd = \text{RANDOM}(start[i], end[i])$

 SWAP($A[i], A[rnd]$)

	1	2	3	4	5	6	7	8	9	10	11
A	10	12	16	25	5	4	8	7	19	6	18
s	1						7				
e	6						11				

Segmented Randomize Pivots

```
procedure SEG-RANDOMIZE-PIVOTS( $A[1..n]$ ,  $\text{segs}[1..n]$ )
```

$start, end$ = new arrays of size n

```
for  $i = 1$  to  $n$  in parallel do
```

```
    if  $\text{segs}[i] = 1$  then  $start[i] = i$ 
```

```
    SEG-BROADCAST( $start[1..n]$ ,  $\text{segs}[1..n]$ )
```

```
for  $i = 1$  to  $n$  in parallel do
```

```
    if  $\text{seg}[i] = 1$  then
```

```
         $rnd = \text{RANDOM}(start[i], end[i])$ 
```

```
        SWAP( $A[i], A[rnd]$ )
```

	1	2	3	4	5	6	7	8	9	10	11
A	10	12	16	25	5	4	8	7	19	6	18
s	1	1	1	1	1	1	7	7	7	7	7
e	6						11				

Segmented Randomize Pivots

```
procedure SEG-RANDOMIZE-PIVOTS( $A[1..n]$ ,  $segs[1..n]$ )
     $start, end$  = new arrays of size  $n$ 
    for  $i = 1$  to  $n$  in parallel do
        if  $segs[i] = 1$  then  $start[i] = i$ 
        SEG-BROADCAST( $start[1..n]$ ,  $segs[1..n]$ )
    for  $i = 1$  to  $n$  in parallel do
        if  $i = n$  OR  $segs[i + 1] = 1$  then  $end[start[i]] = i$ 
    for  $i = 1$  to  $n$  in parallel do
        if  $seg[i] = 1$  then
             $rnd = RANDOM(start[i], end[i])$ 
            SWAP( $A[i], A[rnd]$ )
```

	1	2	3	4	5	6	7	8	9	10	11
A	10	12	16	25	5	4	8	7	19	6	18
s	1	1	1	1	1	1	7	7	7	7	7
e	6						11				

Segmented Randomize Pivots

```
procedure SEG-RANDOMIZE-PIVOTS( $A[1..n]$ ,  $segs[1..n]$ )
     $start, end$  = new arrays of size  $n$ 
    for  $i = 1$  to  $n$  in parallel do
        if  $segs[i] = 1$  then  $start[i] = i$ 
        SEG-BROADCAST( $start[1..n]$ ,  $segs[1..n]$ )
    for  $i = 1$  to  $n$  in parallel do
        if  $i = n$  OR  $segs[i + 1] = 1$  then  $end[start[i]] = i$ 
    for  $i = 1$  to  $n$  in parallel do
        if  $seg[i] = 1$  then
             $rnd = RANDOM(start[i], end[i])$ 
            SWAP( $A[i], A[rnd]$ )
```

	1	2	3	4	5	6	7	8	9	10	11
A	10	12	16	25	5	4	8	7	19	6	18
s	1	1	1	1	1	1	7	7	7	7	7
e	6						11				

Segmented Randomize Pivots

```
procedure SEG-RANDOMIZE-PIVOTS( $A[1..n]$ ,  $\text{segs}[1..n]$ )
     $start, end$  = new arrays of size  $n$ 
    for  $i = 1$  to  $n$  in parallel do
        if  $\text{segs}[i] = 1$  then  $start[i] = i$ 
        SEG-BROADCAST( $start[1..n]$ ,  $\text{segs}[1..n]$ )
    for  $i = 1$  to  $n$  in parallel do
        if  $i = n$  OR  $\text{segs}[i + 1] = 1$  then  $end[start[i]] = i$ 
    for  $i = 1$  to  $n$  in parallel do
        if  $\text{seg}[i] = 1$  then
             $rnd = \text{RANDOM}(start[i], end[i])$ 
            SWAP( $A[i], A[rnd]$ )
```

	1	2	3	4	5	6	7	8	9	10	11
A	10	12	16	25	5	4	8	7	19	6	18
s	1	1	1	1	1	1	7	7	7	7	7
e	6						11				

Segmented Randomize Pivots

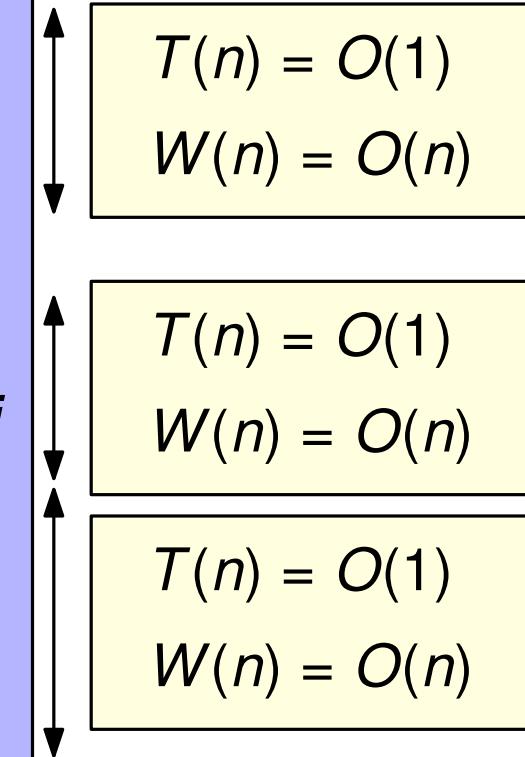
Analysis:

```
procedure SEG-RANDOMIZE-PIVOTS( $A[1..n]$ ,  $segs[1..n]$ )
     $start, end$  = new arrays of size  $n$ 
    for  $i = 1$  to  $n$  in parallel do
        if  $segs[i] = 1$  then  $start[i] = i$ 
        SEG-BROADCAST( $start[1..n]$ ,  $segs[1..n]$ )
    for  $i = 1$  to  $n$  in parallel do
        if  $i = n$  OR  $segs[i + 1] = 1$  then  $end[start[i]] = i$ 
    for  $i = 1$  to  $n$  in parallel do
        if  $seg[i] = 1$  then
             $rnd = RANDOM(start[i], end[i])$ 
            SWAP( $A[i], A[rnd]$ )
```

Segmented Randomize Pivots

Analysis:

```
procedure SEG-RANDOMIZE-PIVOTS( $A[1..n]$ ,  $segs[1..n]$ )
     $start, end$  = new arrays of size  $n$ 
    for  $i = 1$  to  $n$  in parallel do
        if  $segs[i] = 1$  then  $start[i] = i$ 
    SEG-BROADCAST( $start[1..n]$ ,  $segs[1..n]$ )
    for  $i = 1$  to  $n$  in parallel do
        if  $i = n$  OR  $segs[i + 1] = 1$  then  $end[start[i]] = i$ 
    for  $i = 1$  to  $n$  in parallel do
        if  $seg[i] = 1$  then
             $rnd = RANDOM(start[i], end[i])$ 
            SWAP( $A[i], A[rnd]$ )
```



Segmented Randomize Pivots

Analysis:

```
procedure SEG-RANDOMIZE-PIVOTS( $A[1..n]$ ,  $segs[1..n]$ )
     $start, end$  = new arrays of size  $n$ 
    for  $i = 1$  to  $n$  in parallel do
        if  $segs[i] = 1$  then  $start[i] = i$ 
    SEG-BROADCAST( $start[1..n]$ ,  $segs[1..n]$ )
    for  $i = 1$  to  $n$  in parallel do
        if  $i = n$  OR  $segs[i + 1] = 1$  then  $end[start[i]] = i$ 
    for  $i = 1$  to  $n$  in parallel do
        if  $seg[i] = 1$  then
             $rnd = RANDOM(start[i], end[i])$ 
            SWAP( $A[i], A[rnd]$ )
```

Total:

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

Segmented Randomize Pivots

Analysis:

```
procedure SEG-RANDOMIZE-PIVOTS( $A[1..n]$ ,  $segs[1..n]$ )
     $start, end$  = new arrays of size  $n$ 
    for  $i = 1$  to  $n$  in parallel do
        if  $segs[i] = 1$  then  $start[i] = i$ 
    SEG-BROADCAST( $start[1..n]$ ,  $segs[1..n]$ )
    for  $i = 1$  to  $n$  in parallel do
        if  $i = n$  OR  $segs[i + 1] = 1$  then  $end[start[i]] = i$ 
    for  $i = 1$  to  $n$  in parallel do
        if  $seg[i] = 1$  then
             $rnd = RANDOM(start[i], end[i])$ 
            SWAP( $A[i], A[rnd]$ )
```

Total:

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

QUICKSORT($A[1..n]$) Analysis

```
procedure QUICKSORT( $A[1..n]$ )
```

$segs$ = new array of n bits

for $i = 2$ to n **in parallel do** $segs[i] = 0$

$segs[1] = 1$ ▷ initialize segments

while not SORTED($A[1..n]$) **do**

SEG-RANDOMIZE-PIVOTS($A, segs$)

$p[1..n] = \text{SEG-PARTITION}(A[1..n], segs[1..n])$

UPDATE-SEGS($segs[1..n], p[1..n]$)

Analysis:

$O(\log n)$

iterations of the **while** loop in expectation

Each iteration:

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

$$W(n) = O(n)$$

Total:

$$T(n) = O(\log^2 n)$$

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

1 n

10	12	16	25	5	4	19	8
----	----	----	----	---	---	----	---

SEG-PARTITION($A, segs$)

8	5	4	10	12	16	25	19
---	---	---	----	----	----	----	----

$p : 0 \ 0 \ 0 \ 1 \ 0 \ 0 \ 0 \ 0 \ 0$

UPDATE-SEGS($segs, p$)

8	5	4	10	12	16	25	19
---	---	---	----	----	----	----	----

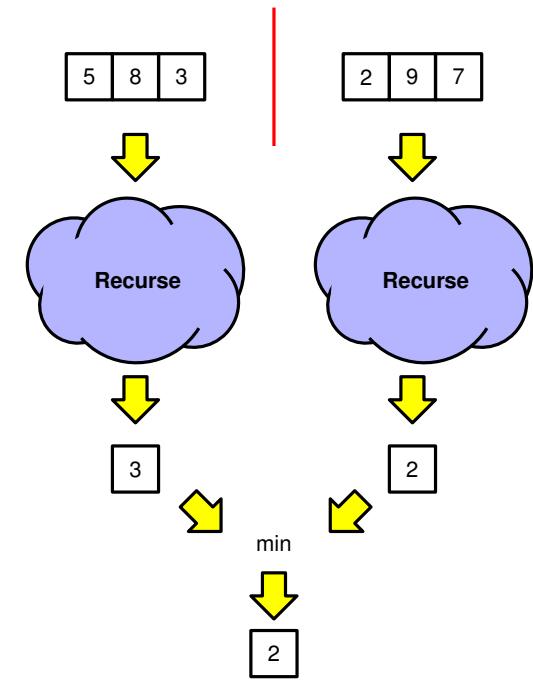


Expected

Finding Minimum

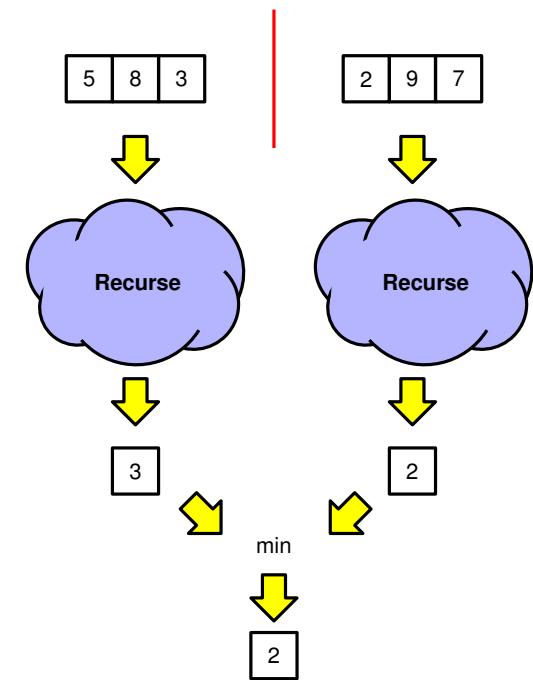
EREW Minimum

```
procedure EREW-MIN( $A[\ell..r]$ )
    if  $\ell = r$  then
        return  $A[\ell]$ 
     $mid = \lfloor \frac{\ell+r}{2} \rfloor$ 
    in parallel do
        left = EREW-MIN( $A[\ell..mid]$ )
        right = EREW-MIN( $A[mid + 1..r]$ )
    return min(left, right)
```



EREW Minimum

```
procedure EREW-MIN( $A[\ell..r]$ )
    if  $\ell = r$  then
        return  $A[\ell]$ 
     $mid = \lfloor \frac{\ell+r}{2} \rfloor$ 
    in parallel do
        left = EREW-MIN( $A[\ell..mid]$ )
        right = EREW-MIN( $A[mid + 1..r]$ )
    return min(left, right)
```



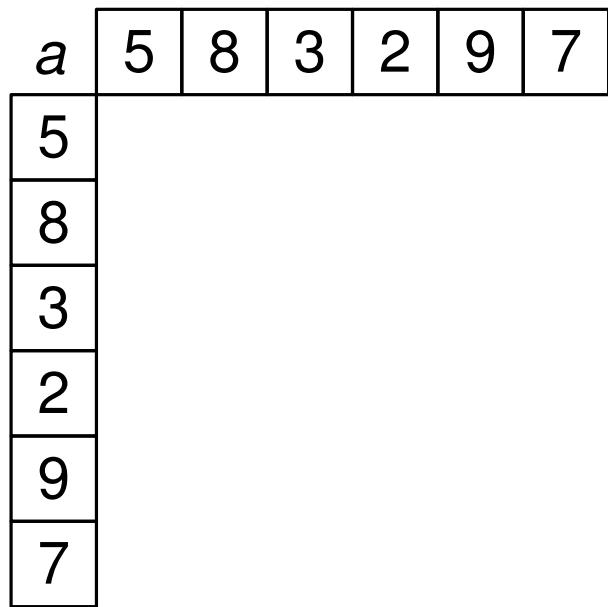
$$T(n) = \begin{cases} T(n/2) + O(1) & \text{if } n > 1 \\ O(1) & \text{if } n = 1 \end{cases} = O(\log n)$$

$$W(n) = \begin{cases} 2W(n/2) + O(1) & \text{if } n > 1 \\ O(1) & \text{if } n = 1 \end{cases} = O(n)$$

Common-CRCW Minimum

a [5 | 8 | 3 | 2 | 9 | 7]

Common-CRCW Minimum



Common-CRCW Minimum

a	5	8	3	2	9	7
5						
8						
3						
2						
9						
7						

M

Common-CRCW Minimum

a	5	8	3	2	9	7
5						
8						
3						
2						
9						
7						

M

$$M[\text{row}, \text{column}] = (a[\text{row}] > a[\text{column}]) ? 1 : 0$$

Common-CRCW Minimum

a	5	8	3	2	9	7
5	0	0	1	1	0	0
8	1	0	1	1	0	1
3	0	0	0	1	0	0
2	0	0	0	0	0	0
9	1	1	1	1	0	1
7	1	0	1	1	0	0

M

$$M[\text{row}, \text{column}] = (a[\text{row}] > a[\text{column}]) ? 1 : 0$$

Common-CRCW Minimum



a	5	8	3	2	9	7
5	0	0	1	1	0	0
8	1	0	1	1	0	1
3	0	0	0	1	0	0
2	0	0	0	0	0	0
9	1	1	1	1	0	1
7	1	0	1	1	0	0

M

$$M[\text{row}, \text{column}] = (a[\text{row}] > a[\text{column}]) ? 1 : 0$$

Common-CRCW Minimum



a	5	8	3	2	9	7
5	0	0	1	1	0	0
8	1	0	1	1	0	1
3	0	0	0	1	0	0
2	0	0	0	0	0	0
9	1	1	1	1	0	1
7	1	0	1	1	0	0

M

$$M[\text{row}, \text{column}] = (a[\text{row}] > a[\text{column}]) ? 1 : 0$$

Common-CRCW Minimum

a	5	8	3	2	9	7
5	0	0	1	1	0	0
8	1	0	1	1	0	1
3	0	0	0	1	0	0
2	0	0	0	0	0	0
9	1	1	1	1	0	1
7	1	0	1	1	0	0

M

```
for  $row = 1$  to  $n$  in parallel do  
  for  $col = 1$  to  $n$  in parallel do  
    if  $a[row] > a[col]$  then  
       $M[row, col] = 1$   
    else  
       $M[row, col] = 0$ 
```

$$M[row, column] = (a[row] > a[column]) ? 1 : 0$$

Common-CRCW Minimum

x	a	5	8	3	2	9	7
0	5	0	0	1	1	0	0
0	8	1	0	1	1	0	1
0	3	0	0	0	1	0	0
0	2	0	0	0	0	0	0
0	9	1	1	1	1	0	1
0	7	1	0	1	1	0	0

M

```
for  $row = 1$  to  $n$  in parallel do  
  for  $col = 1$  to  $n$  in parallel do  
    if  $a[row] > a[col]$  then  
       $M[row, col] = 1$   
    else  
       $M[row, col] = 0$ 
```

$$M[row, column] = (a[row] > a[column]) ? 1 : 0$$

Common-CRCW Minimum

x	a	5	8	3	2	9	7
0	5	0	0	1	1	0	0
0	8	1	0	1	1	0	1
0	3	0	0	0	1	0	0
0	2	0	0	0	0	0	0
0	9	1	1	1	1	0	1
0	7	1	0	1	1	0	0

M

```
for row = 1 to n in parallel do
    for col = 1 to n in parallel do
        if  $a[row] > a[col]$  then
             $M[row, col] = 1$ 
        else
             $M[row, col] = 0$ 
```

```
allocate new array  $x[1..n]$ 
for  $i = 1$  to  $n$  in parallel do
     $x[i] = 0$ 
```

Common-CRCW Minimum

x	a	5	8	3	2	9	7
1	5	0	0	1	1	0	0
1	8	1	0	1	1	0	1
1	3	0	0	0	1	0	0
0	2	0	0	0	0	0	0
1	9	1	1	1	1	0	1
1	7	1	0	1	1	0	0

M

```
for row = 1 to n in parallel do
    for col = 1 to n in parallel do
        if  $a[row] > a[col]$  then
             $M[row, col] = 1$ 
        else
             $M[row, col] = 0$ 
```

```
allocate new array  $x[1..n]$ 
for  $i = 1$  to  $n$  in parallel do
     $x[i] = 0$ 
```

Common-CRCW Minimum

x	a	5	8	3	2	9	7
1	5	0	0	1	1	0	0
1	8	1	0	1	1	0	1
1	3	0	0	0	1	0	0
0	2	0	0	0	0	0	0
1	9	1	1	1	1	0	1
1	7	1	0	1	1	0	0

M

```
for row = 1 to n in parallel do
    for col = 1 to n in parallel do
        if  $a[row] > a[col]$  then
             $M[row, col] = 1$ 
        else
             $M[row, col] = 0$ 
```

```
allocate new array  $x[1..n]$ 
for  $i = 1$  to  $n$  in parallel do
     $x[i] = 0$ 
```

```
for row = 1 to n in parallel do
    for col = 1 to n in parallel do
        if  $M[row, col] == 1$  then
             $x[row] = 1$ 
```

Common-CRCW Minimum

x	a	5	8	3	2	9	7
1	5	0	0	1	1	0	0
1	8	1	0	1	1	0	1
1	3	0	0	0	1	0	0
0	2	0	0	0	0	0	0
1	9	1	1	1	1	0	1
1	7	1	0	1	1	0	0

M

```
for row = 1 to n in parallel do
    for col = 1 to n in parallel do
        if  $a[row] > a[col]$  then
             $M[row, col] = 1$ 
        else
             $M[row, col] = 0$ 
```

```
allocate new array  $x[1..n]$ 
for  $i = 1$  to  $n$  in parallel do
     $x[i] = 0$ 
```

```
for row = 1 to n in parallel do
    for col = 1 to n in parallel do
        if  $M[row, col] == 1$  then
             $x[row] = 1$ 
```

```
for row = 1 to n in parallel do
    if  $x[row] == 0$  then
         $min = a[row]$ 
```

Common-CRCW Minimum

x	a	5	8	3	2	9	7
1	5	0	0	1	1	0	0
1	8	1	0	1	1	0	1
1	3	0	0	0	1	0	0
0	2	0	0	0	0	0	0
1	9	1	1	1	1	0	1
1	7	1	0	1	1	0	0

M

Valid?

```
for row = 1 to n in parallel do
    for col = 1 to n in parallel do
        if  $a[row] > a[col]$  then
             $M[row, col] = 1$ 
        else
             $M[row, col] = 0$ 
```

```
allocate new array  $x[1..n]$ 
for  $i = 1$  to  $n$  in parallel do
     $x[i] = 0$ 
```

```
for row = 1 to n in parallel do
    for col = 1 to n in parallel do
        if  $M[row, col] == 1$  then
             $x[row] = 1$ 
```

```
for row = 1 to n in parallel do
    if  $x[row] == 0$  then
         $min = a[row]$ 
```

Common-CRCW Minimum

x	a	5	8	3	2	9	7
1	5	0	0	1	1	0	0
1	8	1	0	1	1	0	1
1	3	0	0	0	1	0	0
0	2	0	0	0	0	0	0
1	9	1	1	1	1	0	1
1	7	1	0	1	1	0	0

Valid?

M

```
for row = 1 to n in parallel do
    for col = 1 to n in parallel do
        if  $a[row] > a[col]$  then
             $M[row, col] = 1$ 
        else
             $M[row, col] = 0$ 
```

```
allocate new array  $x[1..n]$ 
for  $i = 1$  to  $n$  in parallel do
     $x[i] = 0$ 
```

```
for row = 1 to n in parallel do
    for col = 1 to n in parallel do
        if  $M[row, col] == 1$  then
             $x[row] = 1$ 
```

```
for row = 1 to n in parallel do
    if  $x[row] == 0$  then
         $min = a[row]$ 
```

Common-CRCW Minimum

x	a	5	8	3	2	9	7
1	5	0	0	1	1	0	0
1	8	1	0	1	1	0	1
1	3	0	0	0	1	0	0
0	2	0	0	0	0	0	0
1	9	1	1	1	1	0	1
1	7	1	0	1	1	0	0

M

Analysis:

```
for row = 1 to n in parallel do
    for col = 1 to n in parallel do
        if  $a[row] > a[col]$  then
             $M[row, col] = 1$ 
        else
             $M[row, col] = 0$ 
```

```
allocate new array  $x[1..n]$ 
for  $i = 1$  to  $n$  in parallel do
     $x[i] = 0$ 
```

```
for row = 1 to n in parallel do
    for col = 1 to n in parallel do
        if  $M[row, col] == 1$  then
             $x[row] = 1$ 
```

```
for row = 1 to n in parallel do
    if  $x[row] == 0$  then
         $min = a[row]$ 
```

Common-CRCW Minimum

x	a	5	8	3	2	9	7
1	5	0	0	1	1	0	0
1	8	1	0	1	1	0	1
1	3	0	0	0	1	0	0
0	2	0	0	0	0	0	0
1	9	1	1	1	1	0	1
1	7	1	0	1	1	0	0

M

Analysis:

$$T(n) = O(1)$$
$$W(n) = O(n^2)$$

```
for row = 1 to n in parallel do
    for col = 1 to n in parallel do
        if  $a[row] > a[col]$  then
             $M[row, col] = 1$ 
        else
             $M[row, col] = 0$ 
```

```
allocate new array  $x[1..n]$ 
for  $i = 1$  to  $n$  in parallel do
     $x[i] = 0$ 
```

```
for row = 1 to n in parallel do
    for col = 1 to n in parallel do
        if  $M[row, col] == 1$  then
             $x[row] = 1$ 
```

```
for row = 1 to n in parallel do
    if  $x[row] == 0$  then
         $min = a[row]$ 
```

Common-CRCW Minimum

x	a	5	8	3	2	9	7
1	5	0	0	1	1	0	0
1	8	1	0	1	1	0	1
1	3	0	0	0	1	0	0
0	2	0	0	0	0	0	0
1	9	1	1	1	1	0	1
1	7	1	0	1	1	0	0

Analysis:

$$T(n) = O(1)$$
$$W(n) = O(n^2)$$

procedure FAST-MIN($A[1..n]$)
for $row = 1$ to n **in parallel do**
 for $col = 1$ to n **in parallel do**
 if $a[row] > a[col]$ **then**
 $M[row, col] = 1$
 else
 $M[row, col] = 0$
 allocate new array $x[1..n]$
 for $i = 1$ to n **in parallel do**
 $x[i] = a[i]$
for $row = 1$ to n **in parallel do**
 for $col = 1$ to n **in parallel do**
 if $M[row, col] == 1$ **then**
 $x[row] = 1$
 for $row = 1$ to n **in parallel do**
 if $x[row] == 0$ **then**
 $min = a[row]$

Not work-efficient