Lecture 2-4: More patterns

Advanced Programming for HPC

Professor Lilian Aveneau

1. PARTITION and COMPACT patterns

- SCAN allows to make many new patterns!
- Two examples:
  - PARTITION: move elements to get two sets according to a given predicate
  - COMPACT: same, but keeping only one of the two subsets

1.1 PARTITION (or SPLIT) pattern

- Separate set of values
  - Into two subsets according to a predicate
  - Keeping the respective order of each subset

- Example with $E = \{1,2,3,4,5,6,7,8\}$ and $P = \{0,1,0,1,0,1,0,1\}$
  - Result is $F = \{2,4,6,8,1,3,5,7\}$
  - First the elements validating the predicate,
  - ... then the others

- Algorithm based on 4 steps:
  - MAP, Exclusive SCAN, Inclusive REVERSE SCAN, and PERMUTATION

1.1 PARTITION (or SPLIT) pattern

- First step: initialize two lists of positions (of size $n + 1$).
  - First for leading values, second for trailing values

PARTITION pattern on PRAM CREW machine: first step

```
headPosition[1] ← 0 { for the exclusive SCAN! }
FOR EACH EP i ∈ [1 … p] IN PARALLEL:
    IF predicate[i] THEN
        headPosition[i+1] ← 1 { size n+1 }
        tailPosition[i] ← 0
    ELSE
        headPosition[i+1] ← 0 { size n+1 }
        tailPosition[i] ← 1
    END IF
END FOR
```

1.1 PARTITION (or SPLIT) pattern

- Exclusive SCAN on leading elements (validating predicate)

PARTITION on PRAM CREW machine: step 2, SCAN on the leading elements

```
j ← 1
WHILE j < n:
    FOR each EP i ∈ [1 … n − j] in parallel:
        aux[i] ← headPosition[i]
        aux[i] ← aux[i] ⊕ headPosition[i+j]
        headPosition[i+j] ← aux[i]
    END FOR
    j ← j × 2
END FOR
```

1.1 PARTITION (or SPLIT) pattern

- Inclusive REVERSE SCAN on trailing elements

PARTITION on PRAM CREW machine : step 3, INCLUSIVE REVERSE SCAN

```
j ← 1
WHILE j < n:
    FOR each EP i ∈ [1 ... n − j] in parallel:
        aux[n-(i-1)] ← tailPosition[n-(i-1)]
        aux[n-(i-1)] ←
                aux[n-(i-1)] ⊕ tailPosition[n-(i-1+j)]
        tailPosition[n-(i-1+j)] ← aux[n-(i-1)]
    END FOR
    j ← j × 2
END FOR
```

1.1 PARTITION (or SPLIT) pattern

- Last step: permutation, via a SCATTER

PARTITION on PRAM CREW machine: step 4, permutation

```
{ Last step: SCATTER data to their leading and trailing positions
  and tail positions.
  We need a MAP to know the destinations :
  we calculate it on the fly... }
FOR EACH EP i ∈ [1 ... p] IN PARALLEL:
    IF predicate[i] THEN
        Y[ headPosition[i] + 1 ] ← X[i] { +1 => +Y'First }
    ELSE
        Y[ n - tailPosition[i] + 1 ] ← X[i] {+1 => +Y'First }
    END IF
END FOR
```

1.1 PARTITION (or SPLIT) pattern

Example with {1,2,3,4,5,6,7,8} and predicates {0,1,0,1,0,1,0,1}

- Initialisation :
  - headPosition = {0,0,1,0,1,0,1,0}
  - tailPosition = {1,0,1,0,1,0,1,0}
- Exclusive SCAN (via initialisation !)
  - headPosition = {0,0,1,1,2,2,3,3}
- Inclusive SCAN (left to right)
  - tailPosition = {4,3,3,2,2,1,1,0}
- SCATTER gives
  - $Y = \{2,4,6,8,1,3,5,7\}$

Pattern PARTITION on PRAM CREW machine: first step

PARTITION on PRAM CREW machine: step 2, SCAN over leading elem

PARTITION on PRAM CREW machine: step 4, permutation

```
j←j×2
END IF
END FOR
```

1.2 COMPACT pattern

- Very similar to PARTITION
- Difference
  - Keeps only values whose predicate is True
  - Other values ? Disappear
  - Variable size result $\leq n$
- Other patterns providing "variable" result: ALLOCATE
  - Takes array of values, e.g. $\{2,4,3\}$
  - Provides array of size $2 + 4 + 3$
  - Plus, array (option) index start: $\{0,2,6\}$

1.2 COMPACT pattern

- Algorithm in 4 steps
  - MAP : initialization of the position table
  - SCAN on this array
  - ALLOCATION of the result array (size $\leq n$)
  - SCATTER of the elements whose predicate is true
- SCAN :
  - Exclusive?
    - Good position, but not difficult to do ALLOCATE
  - Inclusive?
    - Bad position, easy ALLOCATE

1.2 COMPACT pattern

- First step: initialize destination positions

COMPACT pattern on a PRAM CREW machine

```
{First initialization step }
FOR EACH EP i ∈ [1 ... p] IN PARALLEL:
    IF predicate[i] THEN
        position[i] ← 1
    ELSE
        position[i] ← 0
    END IF
END FOR
```

1.2 COMPACT pattern

- Second step: Inclusive SCAN

COMPACT pattern on a PRAM CREW machine

```
{ Second step: SCAN on the positions }
j ← 1
WHILE j < n:
    FOR each EP i ∈ [1 … n − j] in parallel:
        { Inclusive SCAN }
        aux[i] ← position[i]
        aux[i] ← aux[i] ⊕ position[i+j]
        position[i+j] ← aux[i]
    END FOR
    j ← j × 2
END FOR
```

1.2 COMPACT pattern

- ALLOCATION : last SCAN value

COMPACT pattern on a PRAM CREW machine

```
{ Allocation }

size ← position[n]
```

1.2 COMPACT pattern

- At last, the **conditional** permutation (or SCATTER_IF)

COMPACT pattern on a PRAM CREW machine

```
{ Last step: conditional SCATTER of the data }
FOR EACH EP i ∈ [1 … p] IN PARALLEL:
    IF predicate[i] THEN
        Y[ position[i] ] ← X[i]
    END IF
END FOR
```

1.2 COMPACT pattern

Example with $\{1,2,3,4,5,6,7,8\}$ and $predicate = IsPrime$

- Initialisation

$$position = \{0,1,1,0,1,0,1,0\}$$

- Inclusive SCAN

$$position = \{0,1,2,2,3,3,4,4\}$$

- ALLOCATE :

$$size = 4$$

- SCATTER_IF

$$Y = \{2,3,5,7\}$$

2.Segmented patterns

Example with SPLIT_AND_MERGE sort

- SCAN, REDUCE, … they are not suitable
- Packets must be processed, and done in parallel...
  - Possible: data are assembled in contiguous arrays
- Know the beginning of each piece (segment)
  - Second array
- Two versions
  - REDUCE
  - SCAN (broad sense)

2.1. SEGMENTED SCAN

- Here inclusive version only
  - But same idea for exclusive, same for reversed ;-)
- Constraint: apply $\oplus$ for segment values only.
- Two inputs:
  - X data.
  - S segments: here, 1 number per X's data.
- Operator is modified to apply to two inputs!

$$\bigoplus\left(\{X_i, S_i\}, \{X_j, S_j\}\right) = \begin{cases} X_i \oplus X_j & if\, S_i = S_j \\ X_j & else \end{cases}$$

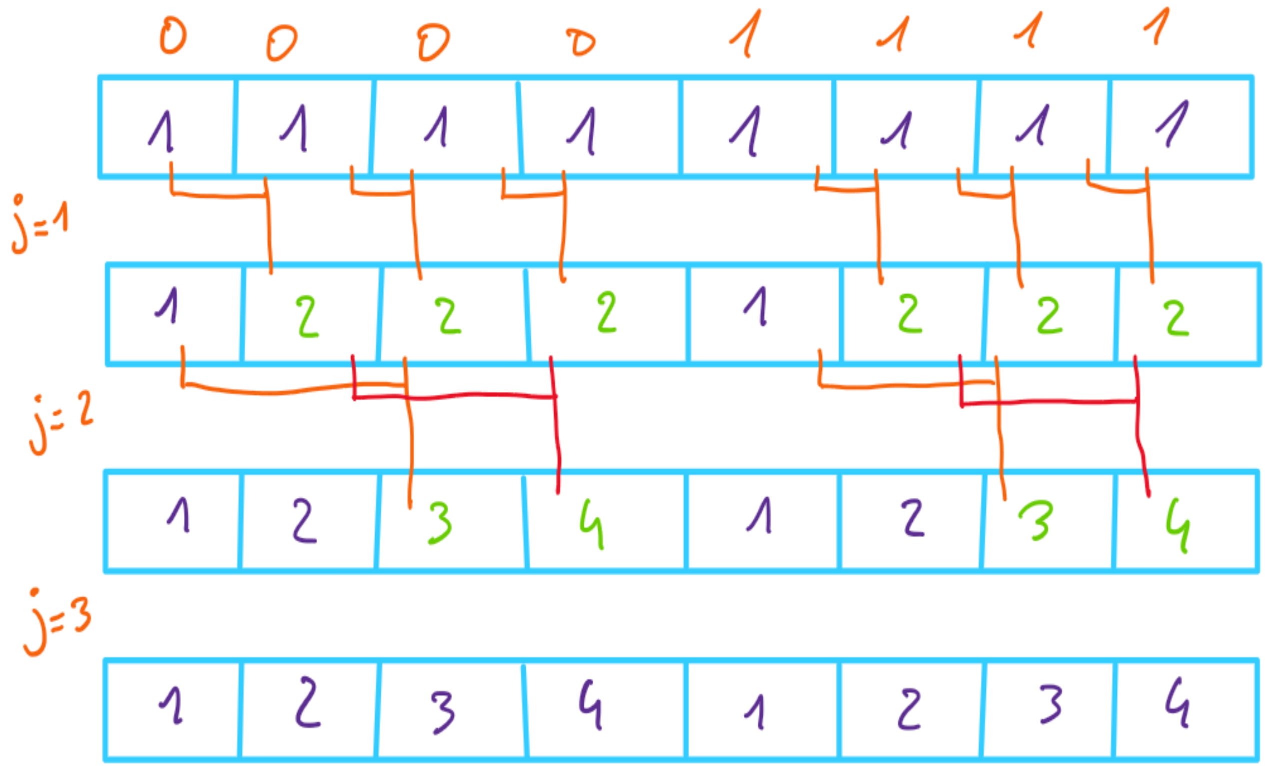
6.1 SEGMENTED SCAN

SEGMENTE inclusive SCAN pattern, EREW version: pointer jump

```
j ← 1
WHILE j < n:
    FOR each EP i ∈ [1 … n − j] in parallel:
        IF S[i] = S[i+j] THEN
            aux[i] ← Y[i]
            aux[i] ← aux[i] ⊕ Y[i+i]
            Y[i+j] ← aux[i]
        END IF
    END FOR
    j ← j × 2
END FOR
```

6.1 SEGMENTED SCAN

- Example with $X = \{1,1,1,1,1,1,1,1\}$ and $S = \{0,0,0,0,1,1,1,1\}$



1.2 SEGMENTED REDUCE

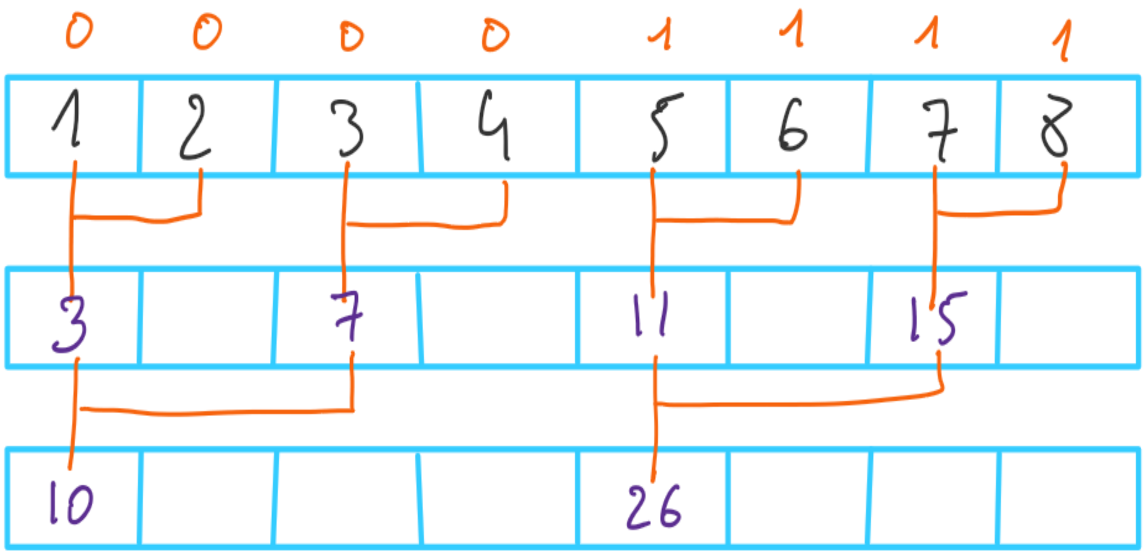
Here, provide a result per segment!

- Example with $X = \{1,2,3,4,5,6,7,8\}$ and $S = \{1,1,1,1,3,3,3,3\}$
- Two segments, therefore two values: $Y = \{10,26\}$
- In PRAM, memory is infinite so no problem
- In practice, it will be necessary to allocate a good size for Y upstream

1.2 SEGMENTED REDUCE

Question: Is the basic version reusable?

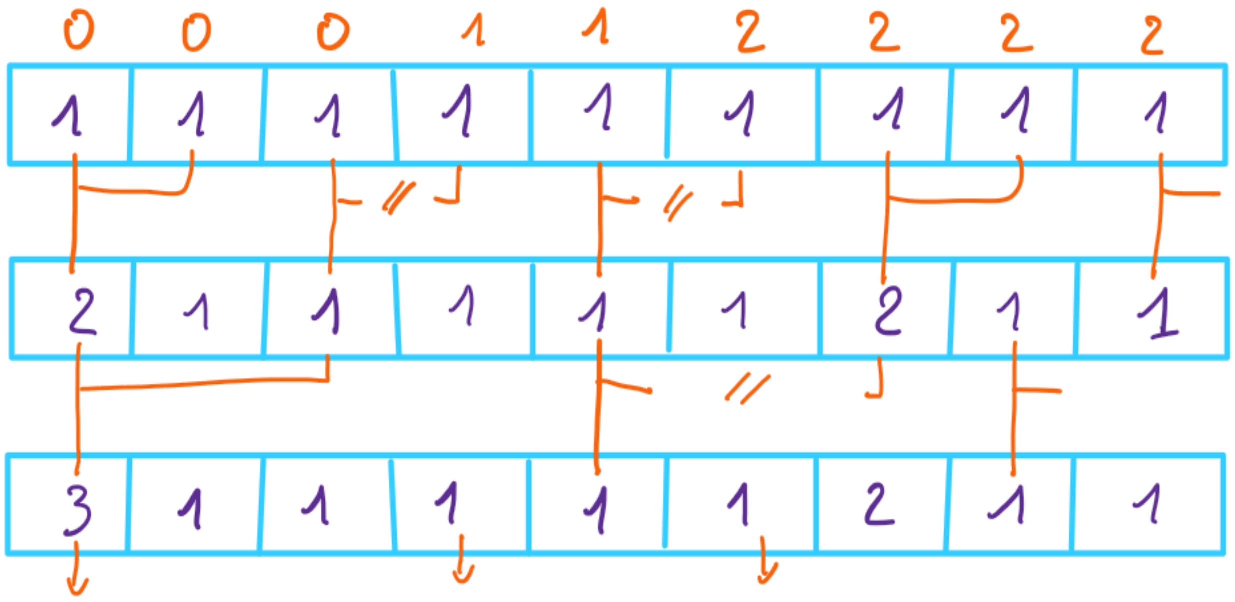
- The one using $n/2$ PE, then $n/4$, then $n/8$ etc.
- Let's try it!



- The last step is unnecessary, otherwise it looks the same...

1.2 SEGMENTED REDUCE

- Let's check again, less regular example $S = \{0,0,0,1,1,2,2,2,2\}$



- Palsembleu! What a wooden pipe! Does not work at all!
- You must use the version seen in the previous chapter ....

1.2 SEGMENTED REDUCE

One step is missing: sending the data to Y!

Several solutions ... one working well:

- For instance, for $S = \{1,1,1,3,3,5,5,5\}$
- The “right" derivative of S, here $D = \{0,0,1,0,1,0,0,1\}$
- Then inclusive SCAN other the derivative: $D = \{0,0,1,1,2,2,2,3\}$
- At last: SCATTER_IF

1.2 SEGMENTED REDUCE

Right derivative is a MAP …

SEGMENTED REDUCE, calculation of the “right“ derivative, EREW mode

```
FOR each EP i ∈ [1 … n − 1] in parallel: { O(1) }
    aux[i] ← S[i+1]
    IF S[i] ≠ aux[i] THEN
        D[i] ← 1
    ELSE
        D[i] ← 0
    END IF
END FOR
D[n] ← 1 { last value is a start of segment }
```

1.2 SEGMENTED REDUCE

Finally, the algorithm becomes

- Inclusive SEGMENTED SCAN: calculation of the sum per segment
- MAP: calculation of the right derivative
- COMPACT!
  - The values: result of the SEGMENTED SCAN
  - The Predicate: right derivative

1.2 SEGMENTED REDUCE

Example

- $S = \{1,1,1,3,3,5,5,5\}$ and $X = \{1,2,3,4,5,6,7,8\}$
- $D = \{0,0,1,0,1,0,0,1\}$ and after scan: $D = \{0,0,1,1,2,2,2,3\}$
- Size of output is last value: 3
- Segmented inclusive scan of X is Y'$= \{1,3,6,4,9,6,13,21\}$
- SCATTER_IF leads to Y'$= \{6,9, 21\}$

3 SORT pattern

Important for many sequential or parallel algorithms

- Better sequential complexity: $O(n \log n)$
- Considering the previous patterns, can we expect $O(\log n)$ in parallel?

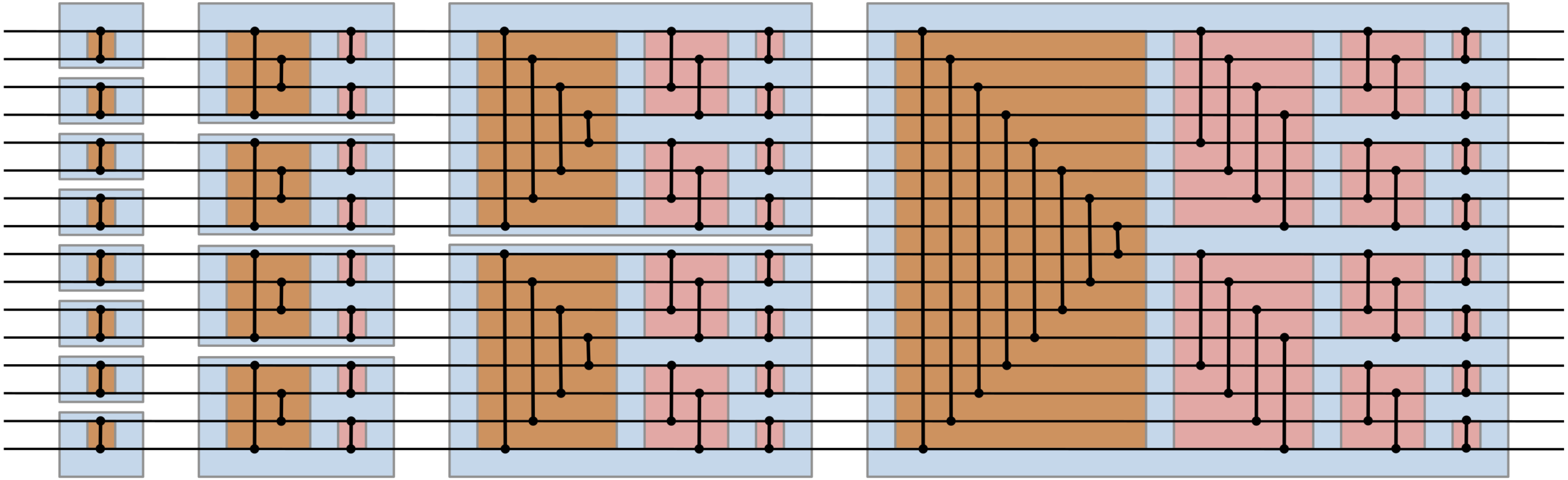
3.1. RADIX SORT

Radix being linked to the number basis (2, 8, 10, 16…)

- Idea: sort by bits, from low to high
- Example: $Y = \{3,1,2,0,2,0,1,3\}$, in 2-basis $\{11,01,10,00,10,00,01,11\}$
- Sorting on bit b_0 (right one)
  - The 0 first, the 1 then… that's a PARTITION!
  - $Y = \{11,01,10,00,10,00,01,11\}$, becomes $Y = \{10,00,10,00,11,01,01,11\}$
- Sorting then on bit b_1 (so the left one)
  - $Y = \{10,00,10,00,11,01,01,11\}$ becomes $Y = \{00,00,01,01,10,10,11,11\}$
- In the end, the table is sorted!

3.2 Sorting based on comparator-exchanger

- Bitonic sorting, from 1968
- Construction of a linear binary sorter network
- Needs $O(\log n)$ **binary comparators**
- Complex but fun architecture (confer English Wikipedia)



3.2 Sorting based on comparator-exchanger

- Simpler version: stack of comparator-exchanger
- Binary function, which orders its inputs (two outputs, therefore)
- Alternation between comparator-exchanger
- $O(n)$ comparators
- Nevertheless, it is interesting
- ... in pipeline mode!