Heterogeneous Parallel Programming COMP4901D

Other Primitives

Overview

- Split
- Sort
  - Bitonic sort
  - Radix sort

Split and Sort

Primitive: Split
Input: $R_{in}[1, \ldots, n]$, $func(R_{in}[i]) \in [1,\ldots,F]$, i=1, …, n.
Output: $R_{out}[1, \ldots, n]$.
Function: $\{R_{out}[i], \ i=1,\ldots, \ n\} = \{R_{in}[i], \ i=1, \ \ldots, \ n\}$ and $func(R_{out}[i]) \le func(R_{out}[j]), \forall i,j \in [1,..,n], i \le j$.

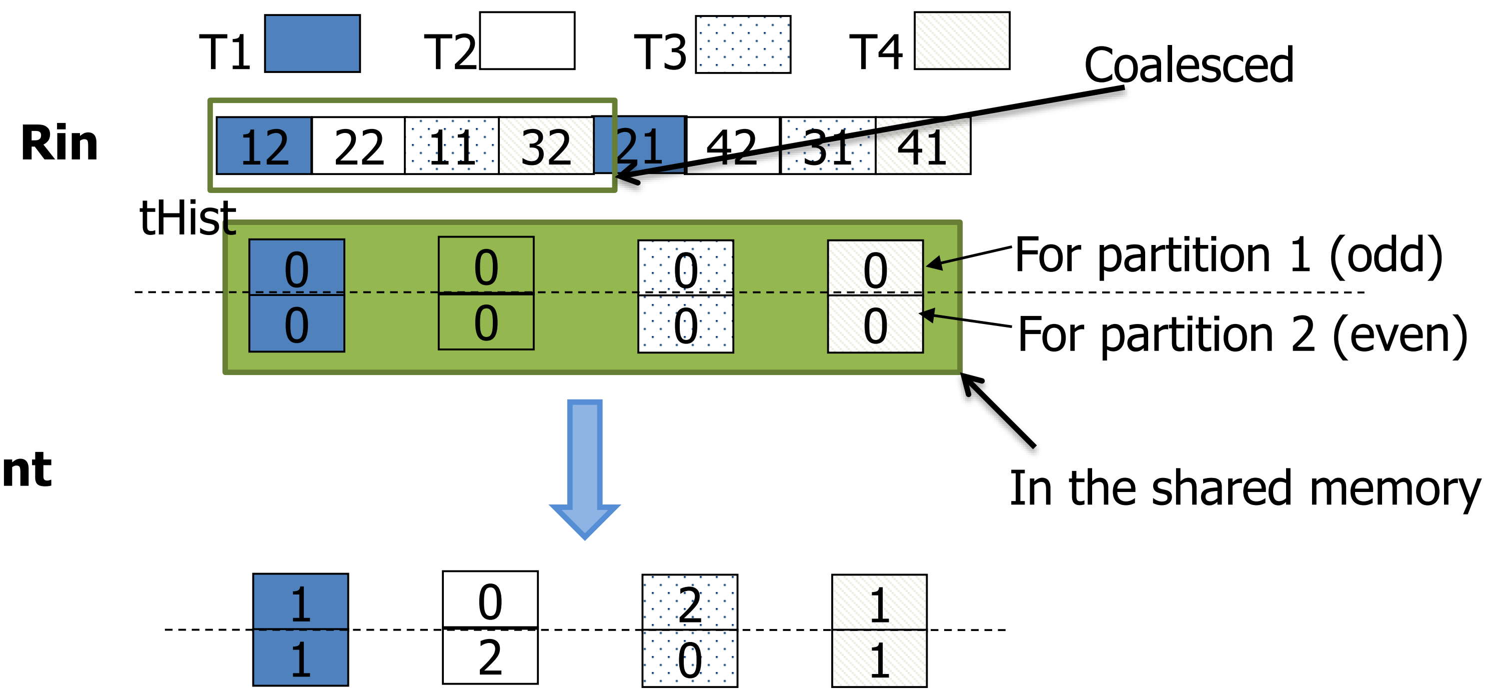
Primitive: Sort
Input: $R_{in}[1, \ldots, n]$.
Output: $R_{out}[1, \ldots, n]$.
Function: $\{R_{out}[i], \ i=1,\ldots, \ n\} = \{R_{in}[i], \ i=1, \ \ldots, \ n\}$ and $R_{out}[i] \le R_{out}[j], \forall i,j \in [1,..,n] \text{ and } i \le j$.

Algorithm for Split

- A lock-free algorithm
  - Each thread is responsible for a portion of the input relation.
  - Each thread computes its local histogram (number of tuples in each output partition).
  - Given the local histograms, each thread computes its write locations.
  - Each thread writes the tuples to the output relation in parallel.

An Example of Split

Split (Rin[1,…, 8], fcn, Rout[1,…,8]), fcn(x)=x mod 2

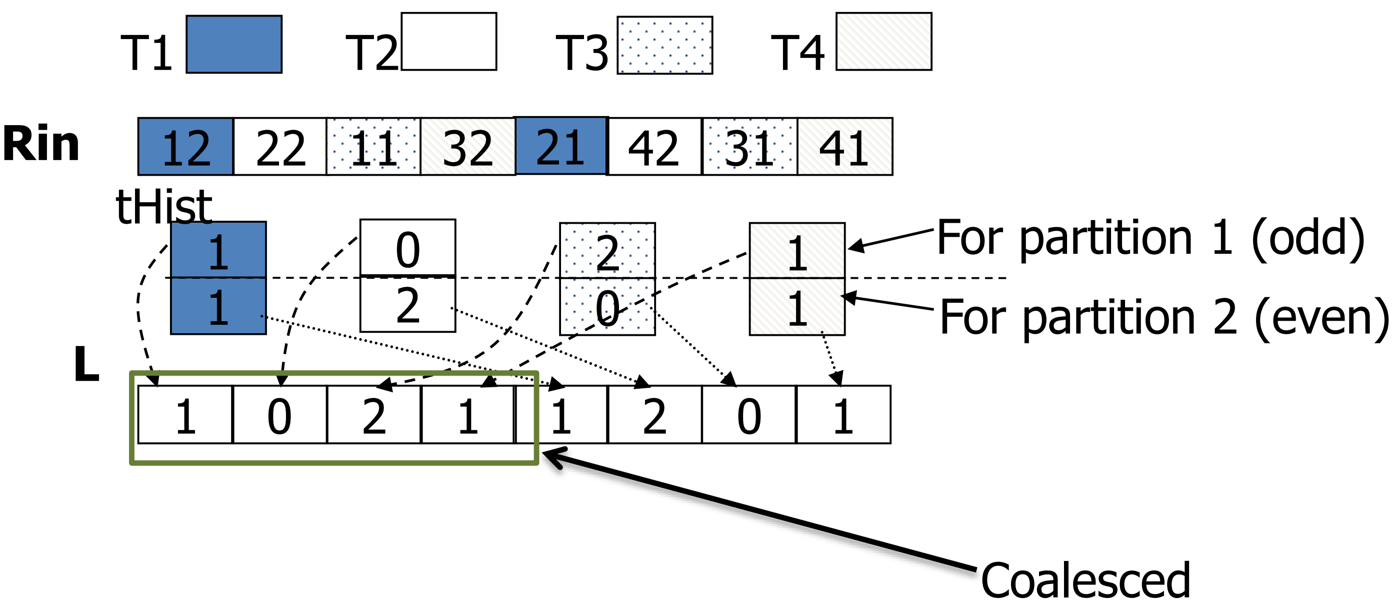


Step (1), count (Map)

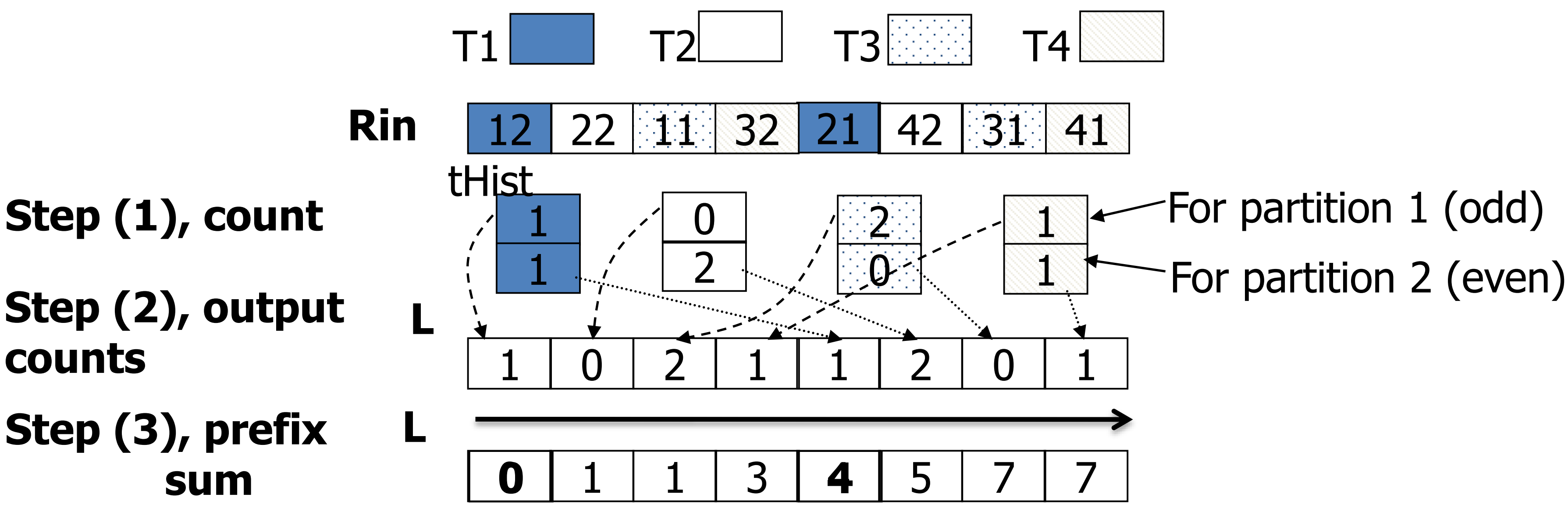
An Example of Split

Step (1), count (Map)

Step (2), output Counts (Scatter)



An Example of Split



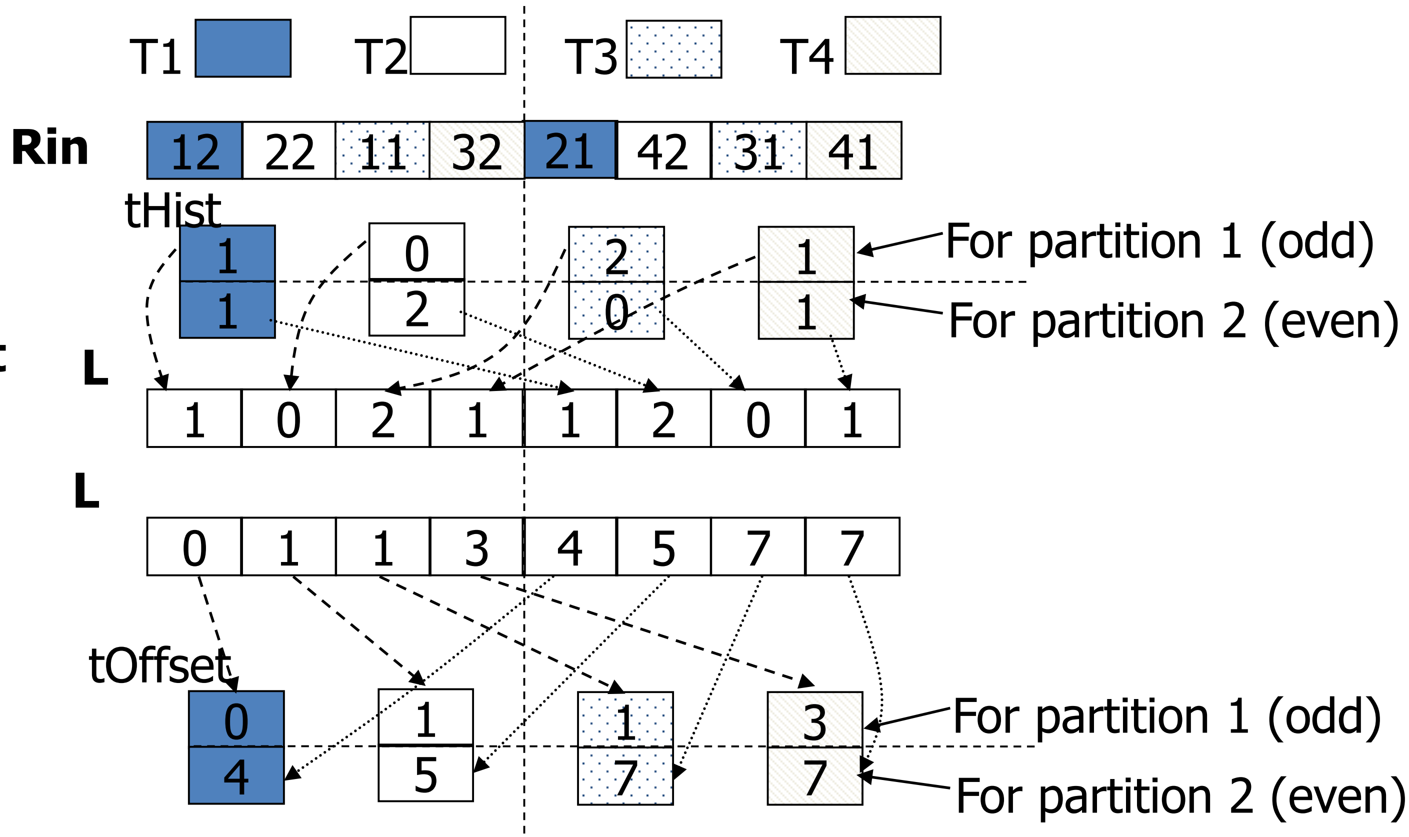
An Example of Split

Step (1), count

Step (2), output counts

Step (3), prefix sum

Step (4), load Counts (Gather)



An Example of Split

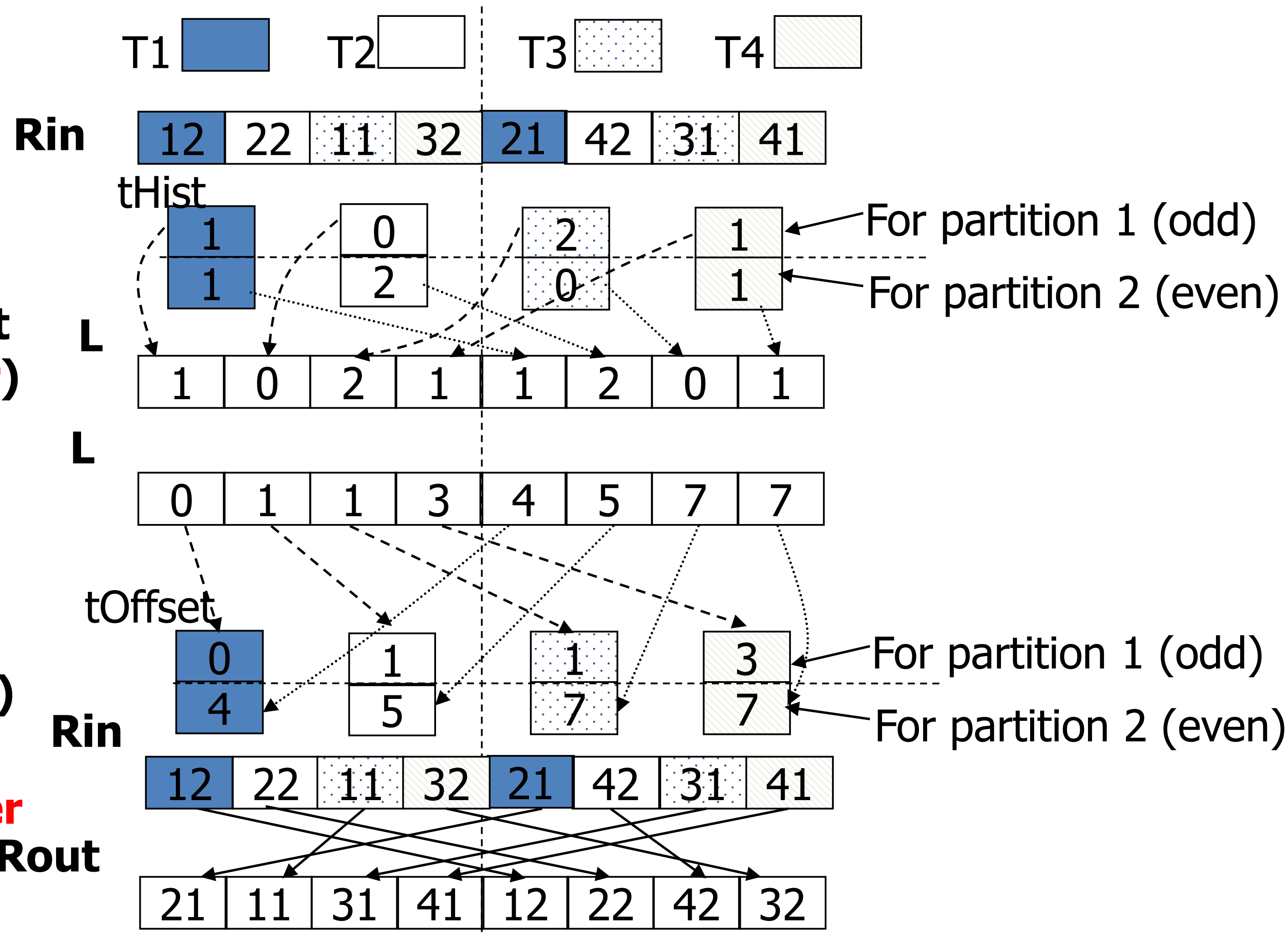
Step (1), count (Map)

Step (2), output Counts (Scatter)

Step (3), prefix sum

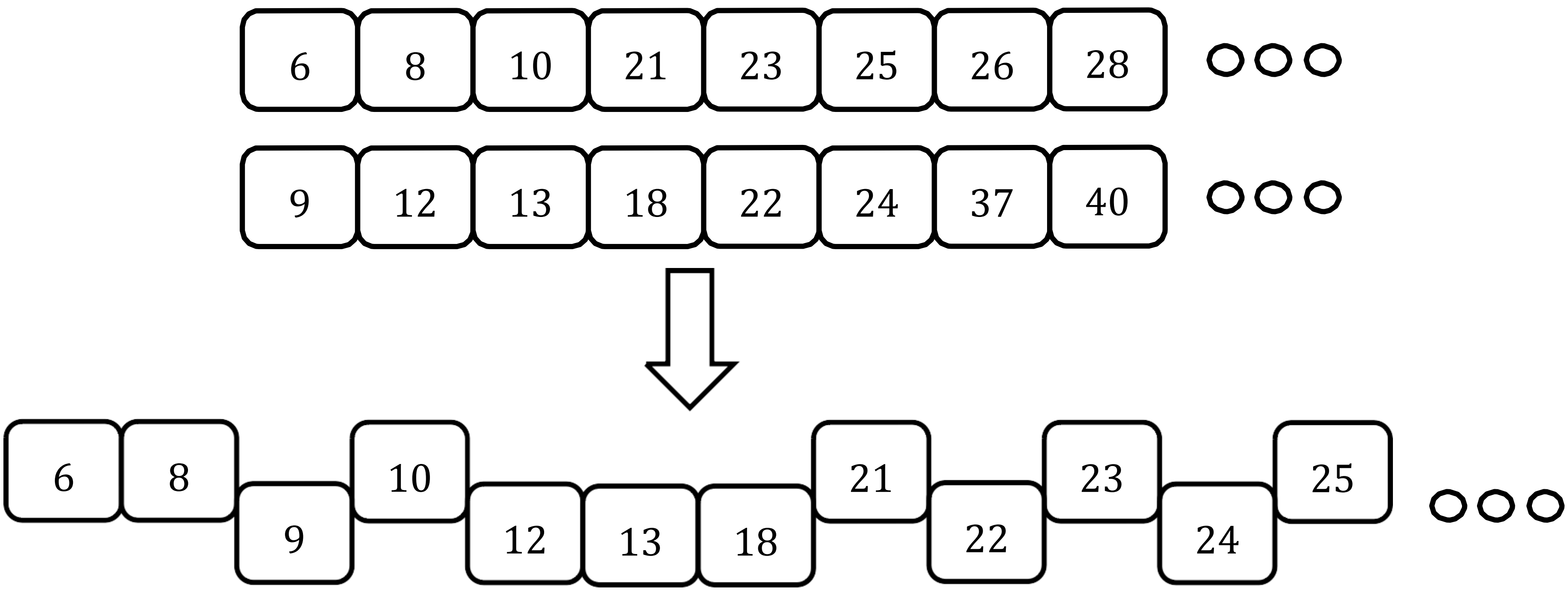
Step (4), load Counts (Gather)

Step (5), scatter



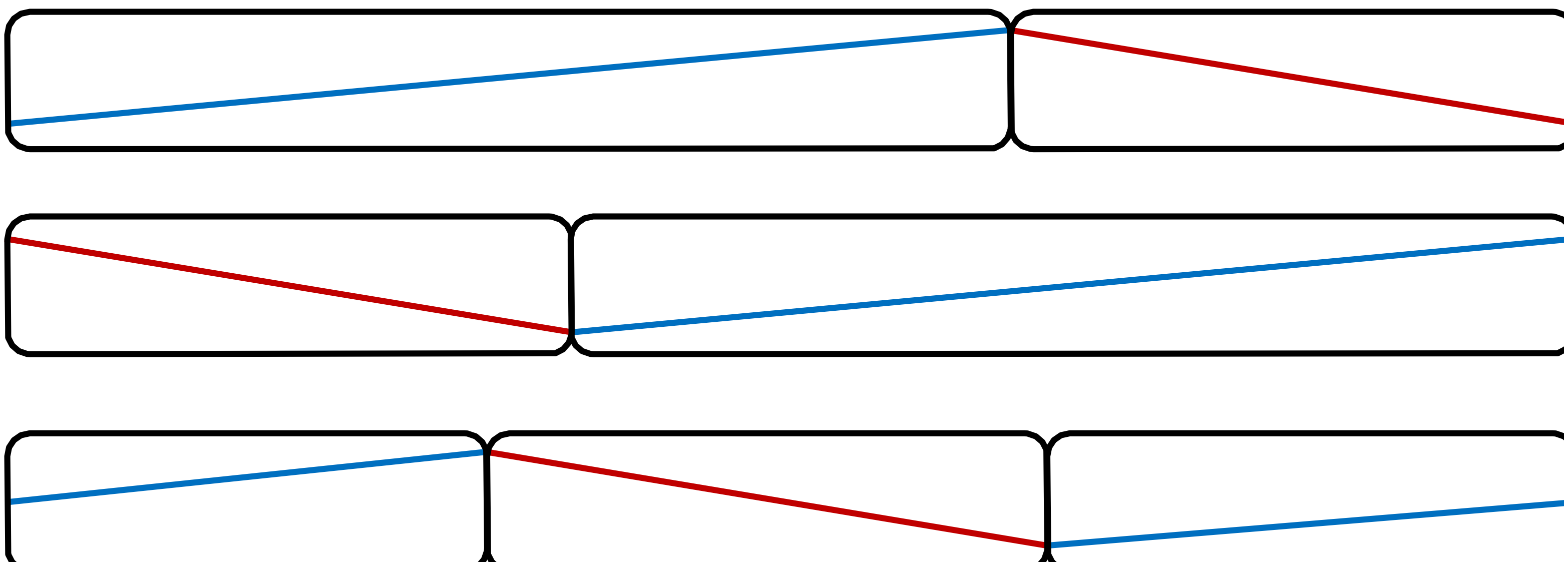
Merge Sort

- Idea: divide and conquer
  - A sequence of size 1 is sorted
  - Merge two sorted sequences into a sorted sequence



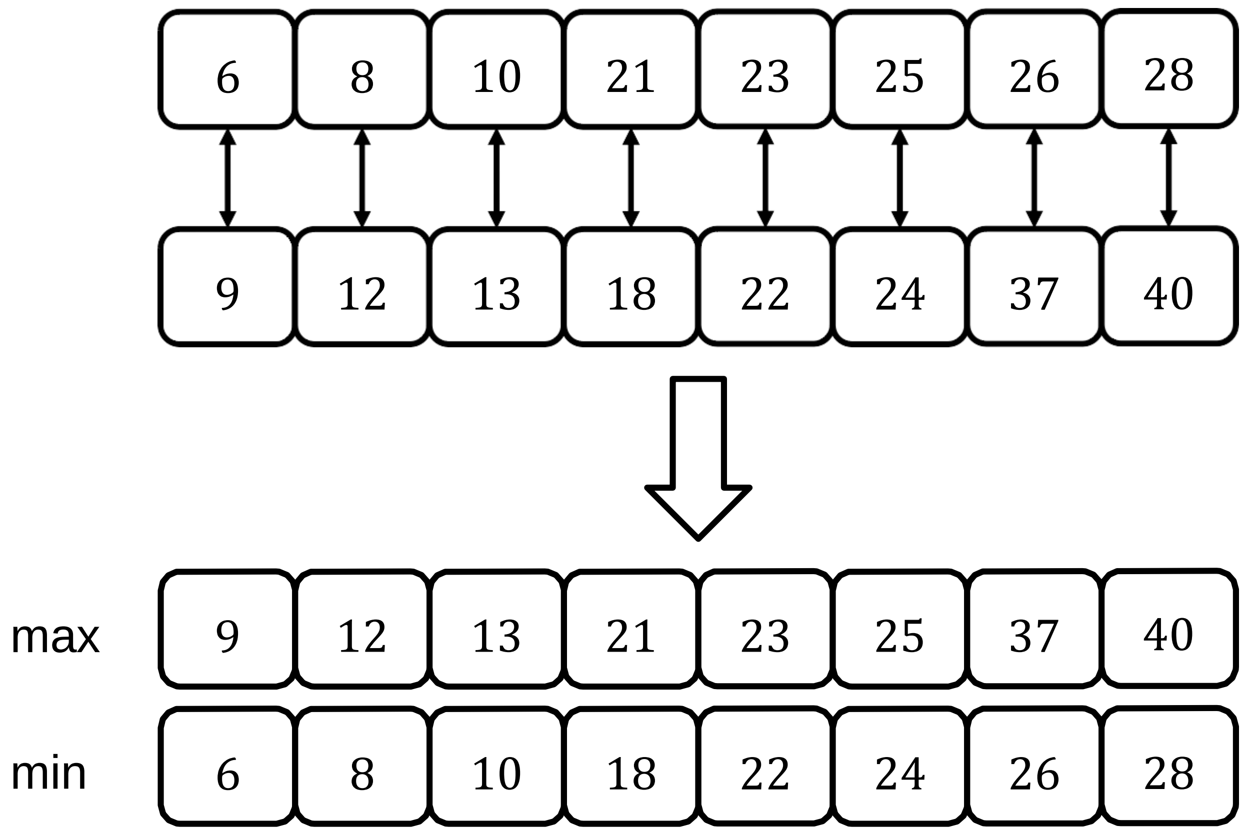
Bitonic Sequence

- A sequence $\boxed{e_1}\boxed{e_2}\boxed{e_3}\boxed{e_4}\boxed{e_5}\boxed{e_6}\boxed{e_7}\boxed{e_8} \cdots \boxed{e_n}$ is bitonic iff
  - Either there is an index i: $0 < i < n$, s.t.
    $\boxed{e_0} \cdots \boxed{e_i}$ is monotonically increasing and
    $\boxed{e_i} \cdots \boxed{e_n}$ is monotonically decreasing
  - Or there is a cyclic shift of the sequence, for which the above holds



Comparison Network

- An array of “processors” performing element-wise comparison of two input sequences



Bitonic Split

- A bitonic split divides a **bitonic** sequence in two through a comparison network

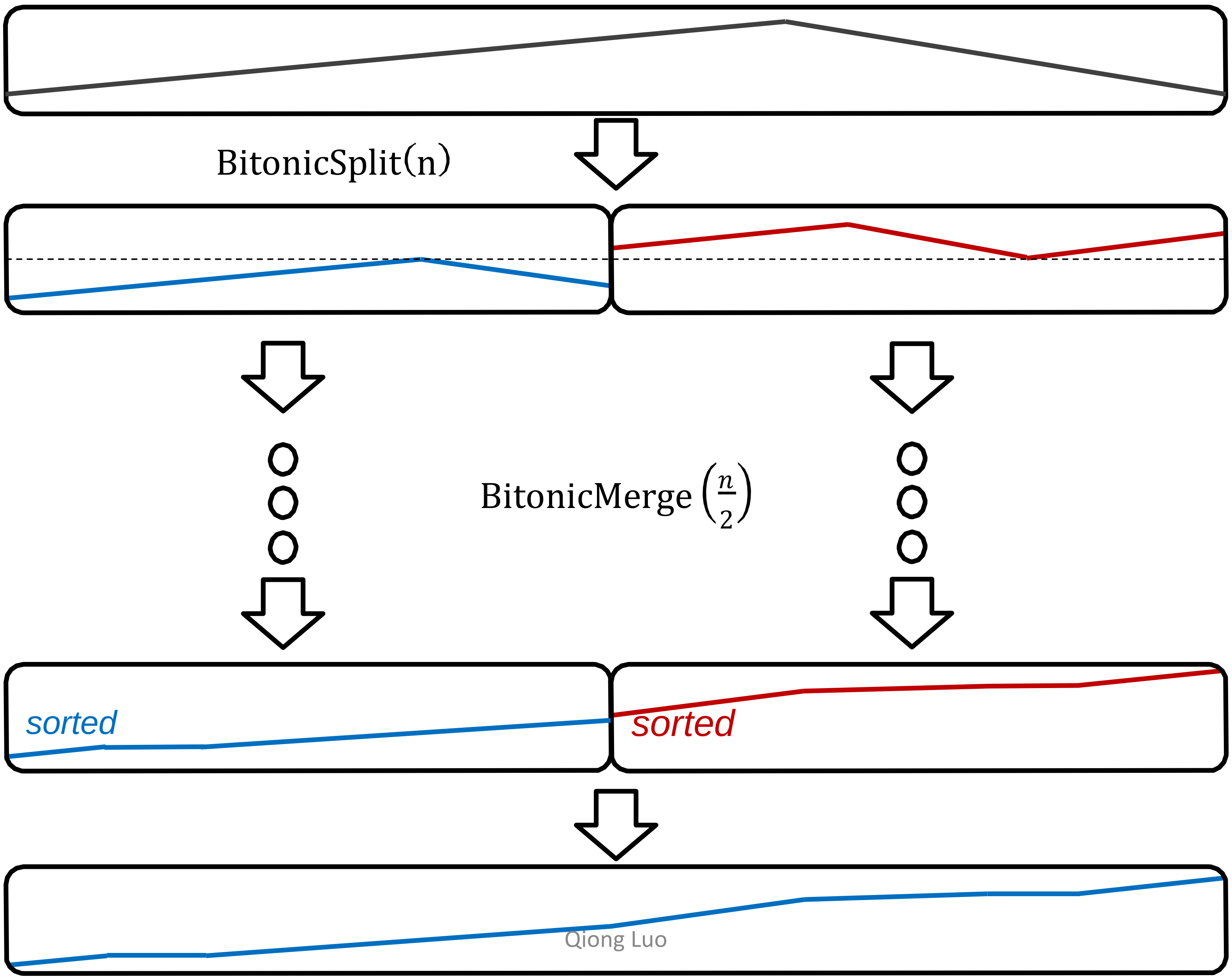
```
BitonicSplit(bs[2*N])
```

⇩

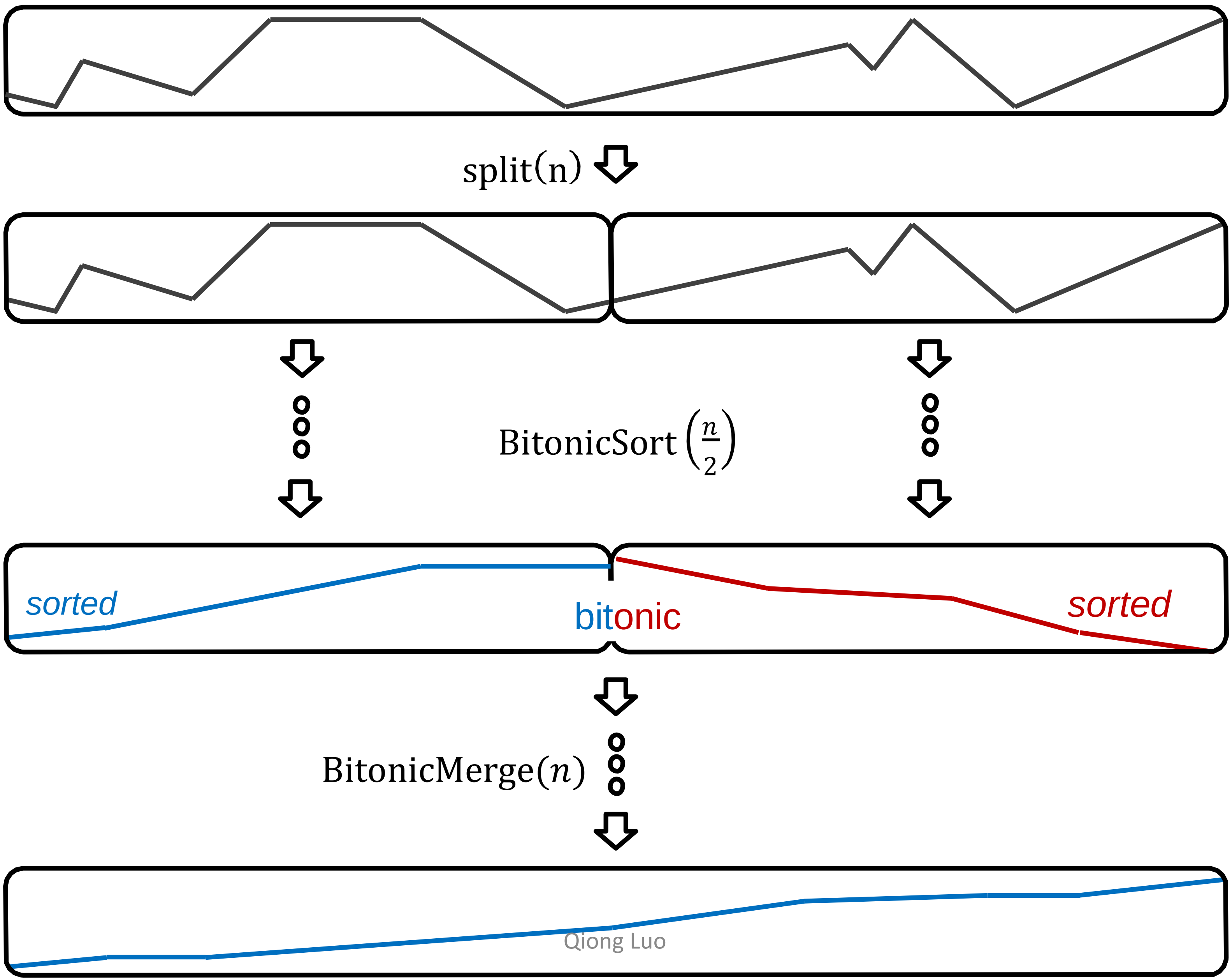
```
S1[N] = { min( bs[0], bs[N] ), … , min( bs[N-1], bs[2*N-1] ) }
S2[N] = { max( bs[0], bs[N] ), … , max( bs[N-1], bs[2*N-1] ) }
```

- Theorem
  - S1, S2 both bitonic
  - S1 < S2

Bitonic Merge(n)



Bitonic Sort(n)



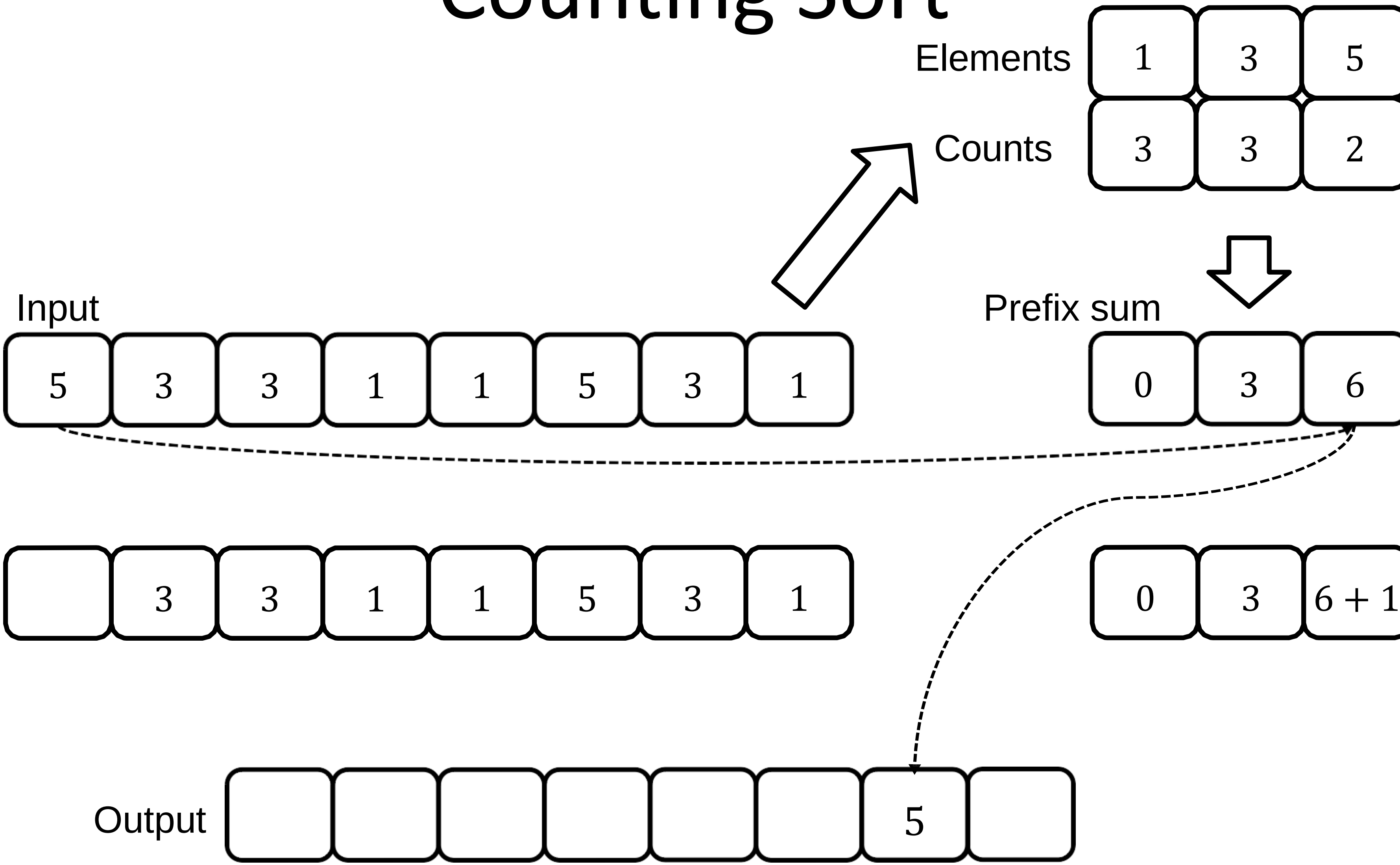
Bitonic Sort Summary

- Work - $O(n \log^2 n)$
  time – $O(\log^2 n)$
- Fixed comparison networks
  - Full hardware utilization at each time step

```
shared[tid] = values[tid]; // Copy input to shared memory
__syncthreads();

for (int k = 2; k <= NUM; k *= 2) //Parallel bitonic sort
{
  for (int j = k / 2; j>0; j /= 2) //Bitonic merge
  {
    int ixj = tid ^ j; //XOR
    if (ixj > tid)
      if ((tid & k) == 0) // ascending – descending
      {
        if (shared[tid] > shared[ixj])
          swap(shared[tid], shared[ixj]);
      }
      else
      {
        if (shared[tid] < shared[ixj])
          swap(shared[tid], shared[ixj]);
      }
    __syncthreads();
  }
}
values[tid] = shared[tid]; // Write result
```

Counting Sort



Counting Sort (2)

- Integer sorting algorithm (no comparisons)
- Complexity (sequential) - $O(n)$
- Assume
  - Constant number of possible values
  - Possible values known in advance
- Algorithm outline
  - Count the occurrences of each element
    - Histogram
  - For each element compute its index in the sorted array
    - Prefix sum over the histogram
  - Move each element to its location

Radix Sort

- Iterated counting sort on individual digits (radix)
- Sort from the least significant to the most significant
  - Use (stable) counting sort

Input	25	13	53	21	61	45	83	11
Sort 2nd digit	21	61	11	13	53	83	25	45
Sort 1st digit	11	13	21	25	45	53	61	83

Radix Sort Analysis

- Integer sort
- Complexity $\mathrm{O}(\mathrm{k}n)$
  - n – number of elements
  - k– number of digits (constant)
- Parallel implementation (naïve)
  - For each radix(digit)
    - Compute radix histogram
    - Scan the histogram to compute offset for each element
    - Write the sorted elements
  - Work $\mathrm{O}(\mathrm{k}n)$
  - Time $\mathrm{O}(\mathrm{k}\log n)$ (because of the global prefix sum)

Radix Sort & CUDA

- Currently the fastest GPU sort algorithm
  - Much faster than any sort on a single CPU (in 2009)
- Works for integers and floats
- Fastest implementation by back40computing
  - http://code.google.com/p/back40computing/wiki/RadixSorting
  - Also used in Thrust
- Another very fast implementation by Markus Billeter
  - http://www.cse.chalmers.se/~billeter/pub/pp/index.html
  - Based on efficient stream compaction

Algorithms for Radix Sort

- Features
  - Efficient scatter
  - Shared memory optimization
- Two steps
  - Divide the input data into multiple partitions.
    - Histogram-based computation to figure out the output position of each element.
    - Histograms are stored in the shared memory.
    - Scatter the output.
  - Sort the partitions in parallel using bitonic sort.
    - The bitonic sort in the shared memory is efficient.

Summary

- Split and Sort are two other data-parallel primitives.
- They can be composed using simpler primitives.
- They are used widely in higher-level applications.