Parallel Sorting

A jungle

Exchange sorts	Bubble sort · Cocktail sort · Odd–even sort · Comb sort · Gnome sort · Quicksort · Stooge sort · Bogosort
Selection sorts	Selection sort · Heapsort · Smoothsort · Cartesian tree sort · Tournament sort · Cycle sort
Insertion sorts	Insertion sort · Shellsort · Splaysort · Tree sort · Library sort · Patience sorting
Merge sorts	Merge sort · Cascade merge sort · Oscillating merge sort · Polyphase merge sort · Strand sort
Distribution sorts	American flag sort · Bead sort · Bucket sort · Burstsort · Counting sort · Pigeonhole sort · Proxmap sort · Radix sort · Flashsort
Concurrent sorts	**Bitonic sorter** · Batcher odd–even mergesort · Pairwise sorting network
Hybrid sorts	Block sort · Timsort · Introsort · Spreadsort · JSort
Other	Topological sorting · Pancake sorting · Spaghetti sort

Illustration

https://www.youtube.com/watch?v=kPRA0W1kECg

(Sequential) Sorting

- Bubble Sort, Insertion Sort
  - $O(n^2)$
- Merge Sort, Heap Sort, QuickSort
  - $O(n \log n)$
  - QuickSort best on average
- Optimal Parallel Time complexity
  - $O(n \log n) / P$
  - If $P = N$ then $O(\log n)$

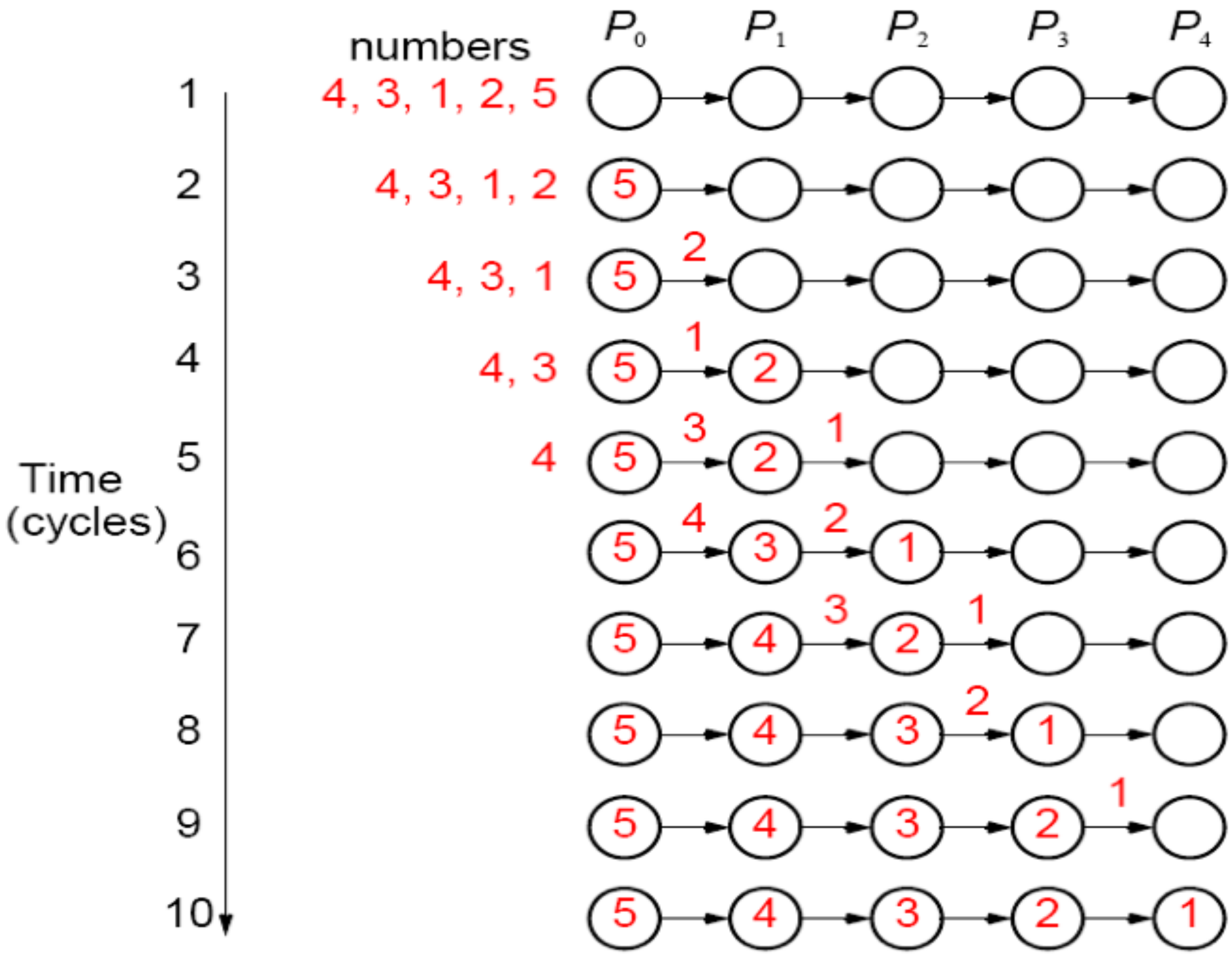
Insertion Sort

```
Insertion_Sort (A)
   for i from 1 to |A| - 1
     j = i
     while j > 0 and A[j-1] > A[j]
          swap A[j] and A[j-1]
          j = j - 1
Return ( A )
```

Inherently sequential so hard to parallelize !!!!

➔ Only through pipelining can speedup be realized

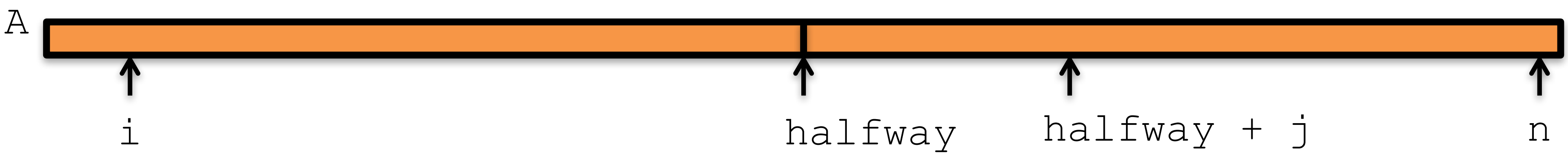
Pipelined Insertion Sort



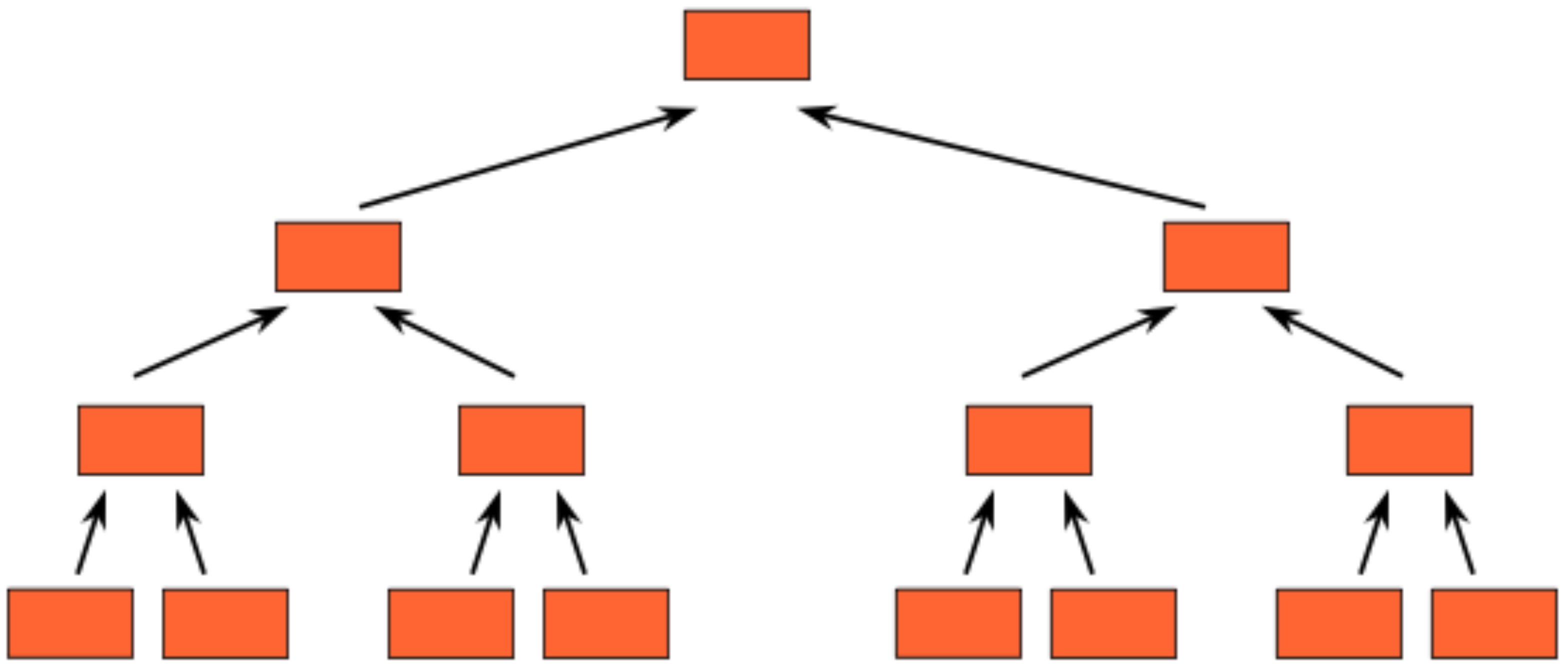
$T_{pipelined}$ = 2n, with n processors, so maximal speedup = n/4 – 3/4 (wortcase sequential time = (n-1)(n-2)/2 = n^2/2-3n/2+2/2)

Parallel Merge Sort

```
Merge_Sort (A)
   n = |A|
   halfway = floor(n/2)

   DO IN PARALLEL
      Merge_Sort (A[1]… A[halfway])
      Merge_Sort (A[halfway+1]… A[n])

   j = 1; current = 1
   for i from 1 to halfway
      while j ≤ n-halfway and A[halfway + j] < A[i]
        X[current] = A[halfway + j]
        j = j + 1; current = current+1
      X[current] = A[i]
      current = current+1
Return ( X )
```



In a picture



Notes Merge Sort

- Collects sorted list onto one processor, merging as items come together
- Maps well to tree structure, sorting locally on leaves, then merging up the tree
- As items approach root of tree, processors are dropped, limiting parallelism
- O (n), if P = n

$$(1+2+4+\ldots+n/2+n) = n\,(1+1/2+1/4\ \ldots\ ) = n.2$$

Parallel QuickSort

```
QuickSort (A)
    if |A| == 1 then return A
    i = rand_int (|A|)
    p = A[i]
                              ∈
    DO IN PARALLEL
        L = QuickSort({a∈A|a < p})
        E = {a∈A|a = p}
        G = QuickSort({a∈A|a > p})
Return ( L || E || G )
```

If we assume that the pivots are chosen such that L and G are about equal in size, then

$$\text{Sequential: } T(n) = 2T(n/2) + O(n) = O(n \log n)$$

In fact it can be proven that this always holds!

For parallel execution the choice of `i` is crucial for load balance. Even more importantly we would like to choose multiple pivots (p-1) at the same time, so that each time we get p partitions which can be executed in parallel.

P partitions

- For a given p (number of pivots) and s (oversampling rate), first select at random

  p*s candidate pivots

```
for i from 1 to p*s
   Cand[i] = rand_int (|A|)
```

- Sort the list of candidate pivots: `Cand[i]`
- Choose `Cand[s],Cand[2*s]…Cand[(p-1)*s]`

Find a good value for the oversampling rate: s > 1,

➔ s should not lead to very long sorting times

Parallel Radix Sort

Instead of comparing values: **COMPARE DIGITS**

```
Radix_Sort (A, b)  # Assume binary representations of keys
   for i from 0 to b-1
      FLAGS = { (a>>i) mod 2 | a∈A }
      NOTFLAGS = { 1-FLAGS[a] | a∈A }
      R_0 = SCAN (NOTFLAGS)
      s_0 = SUM (NOTFLAGS)
      R_1 = SCAN (FLAGS)
      R = {if FLAGS[j] == 0
            then R_0[j]
            else R_1[j] + s_0
           | j∈[0…|A|-1}
      A = A sorted by R
Return ( A )
```

`(a>>i) mod 2`:

rightshift `i` times, so e.g.

`01101>>2 mod2 =`

`00011 mod 2 = 1`

So `(a>>i) mod 2` equals the `(i+1)`th rightmost bit of a

LSD/MSD Radix Sort

Instead of

```
(a>>i) mod 2
```

one can also implements Radix Sort with:

```
(a<<i) div 2^(b-1)
```

The first implementation is called least significant digit Radix Sort or LSD Radix Sort

The latter on is MSD Radix Sort

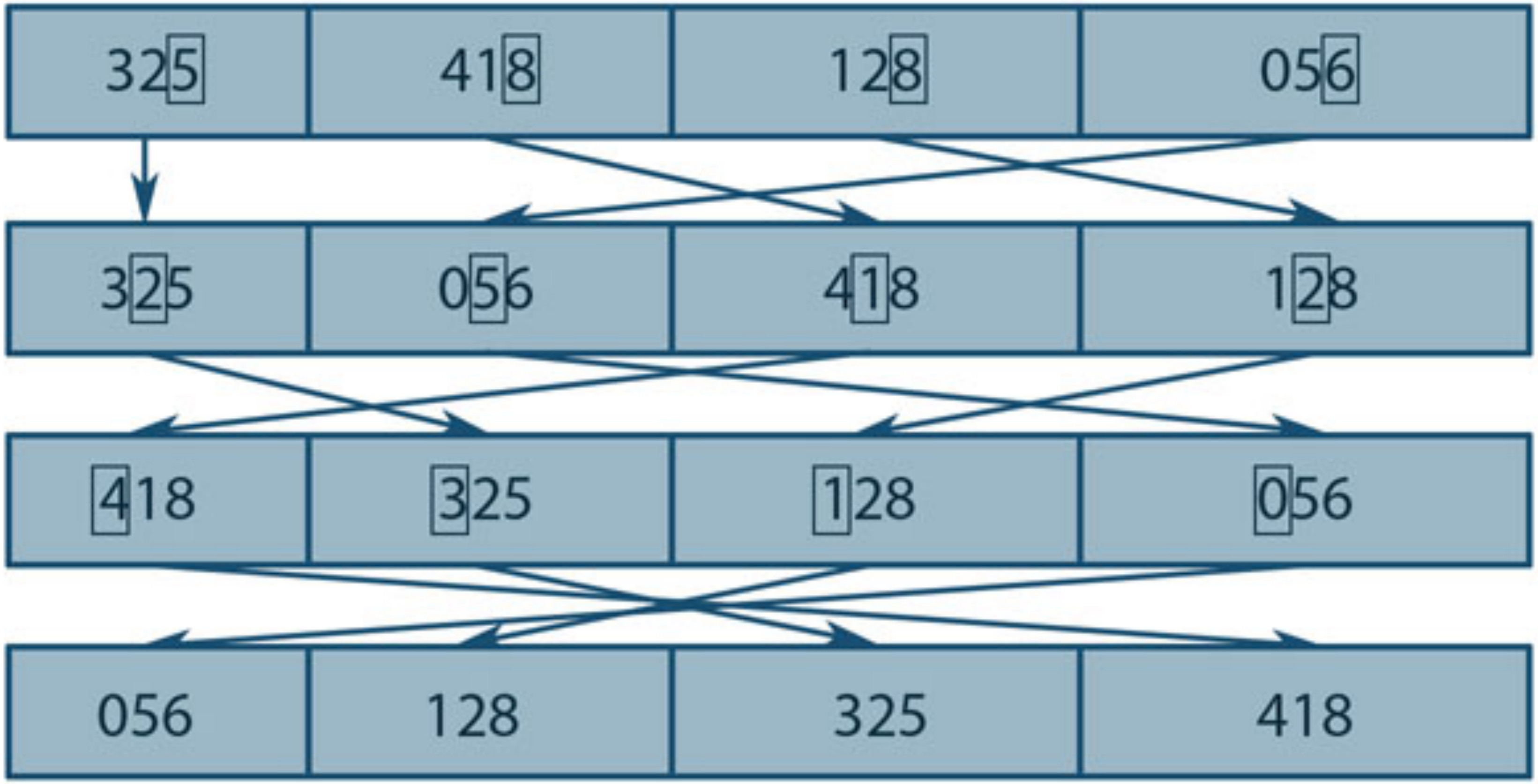
Notes Radix Sort

- Sequential time complexity:

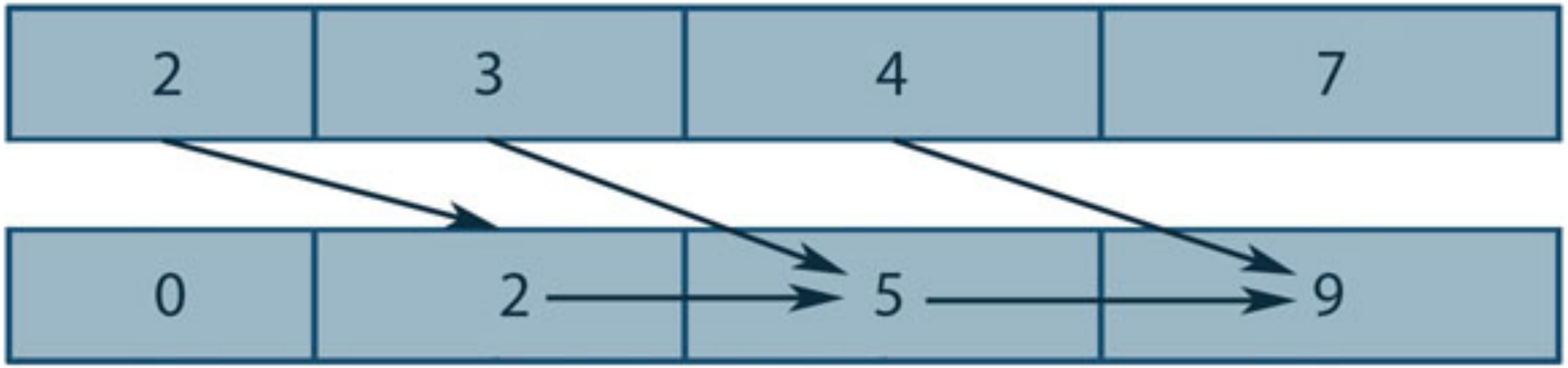
  $T(n) = O\ (b.n)$,

  b iterations, each iteration $O(n)$
- Note that $b \approx \log n$, so a total of $O\ (n \log n)$
- Instead of single digits a block of r digits can be taken each time, resulting in b/r iterations

Illustration (LSD Radix Sort)

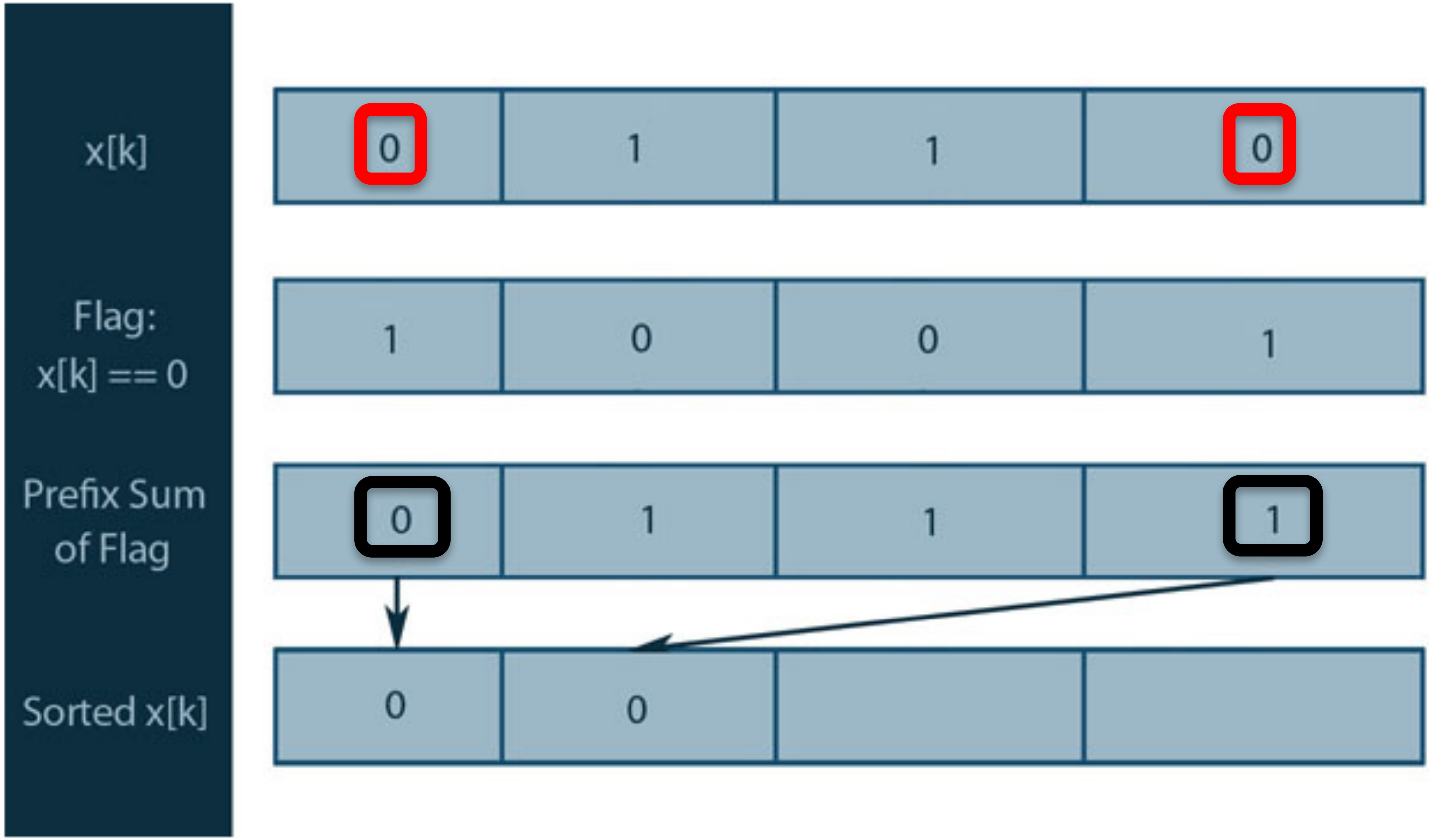


Sorting of each selected digit in Radix Sort, with Prefix Sum Based Sorting



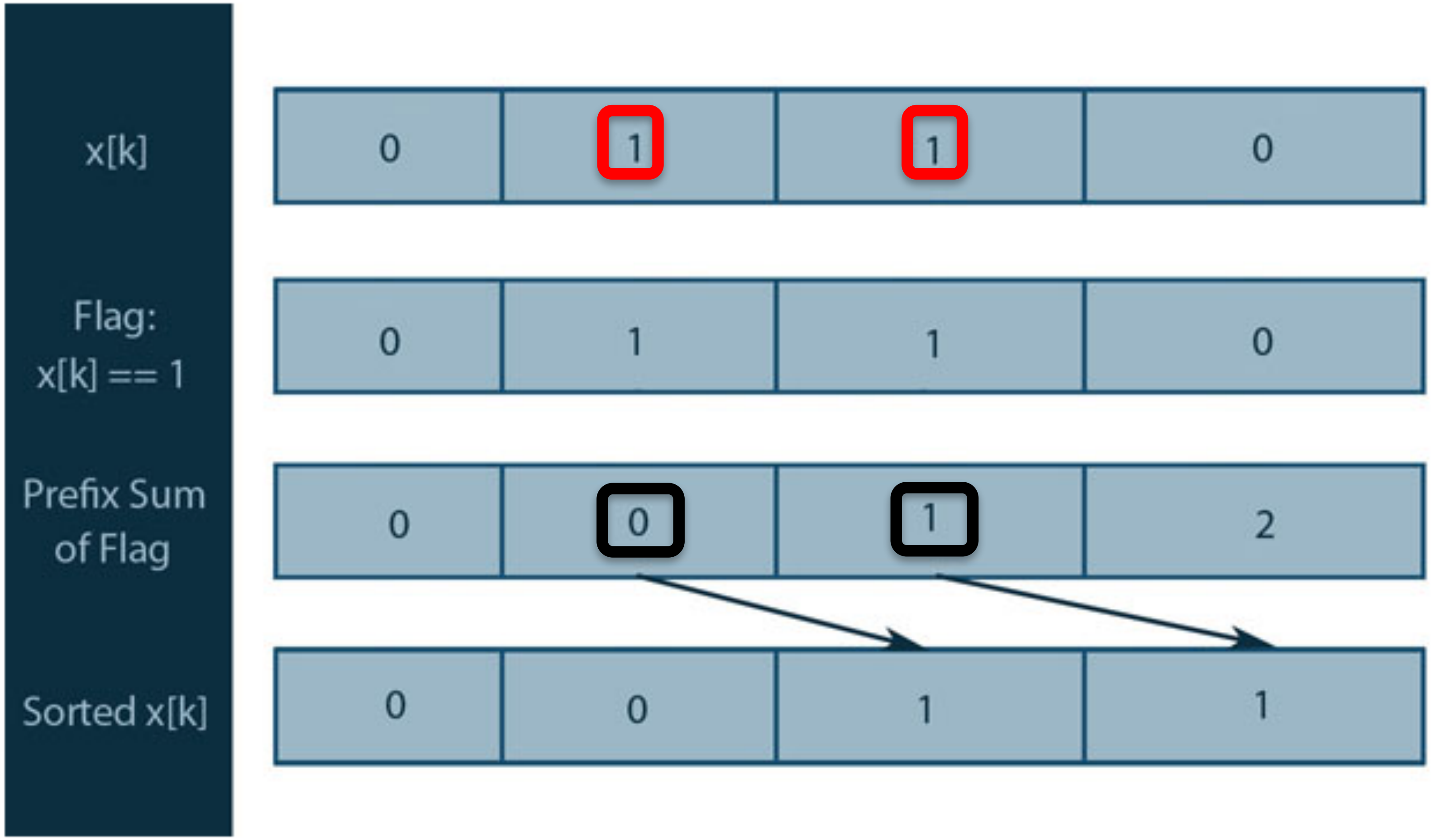
Each element i of the prefix sum array has the SUM of all elements which index is smaller than i

What is the relationship with sorting?



- All bits which are equal to 0 are flagged with a 1
- Compute Prefix Sum of this flag array
- Store all flagged (1) entries of x[k] in the location indicated by the prefix sum

Second stage



- All bits which are equal to 1 are flagged with a 1
- Compute Prefix Sum of this flag array
- Store all flagged (1) entries of x[k] in the next locations indicated by the prefix sum

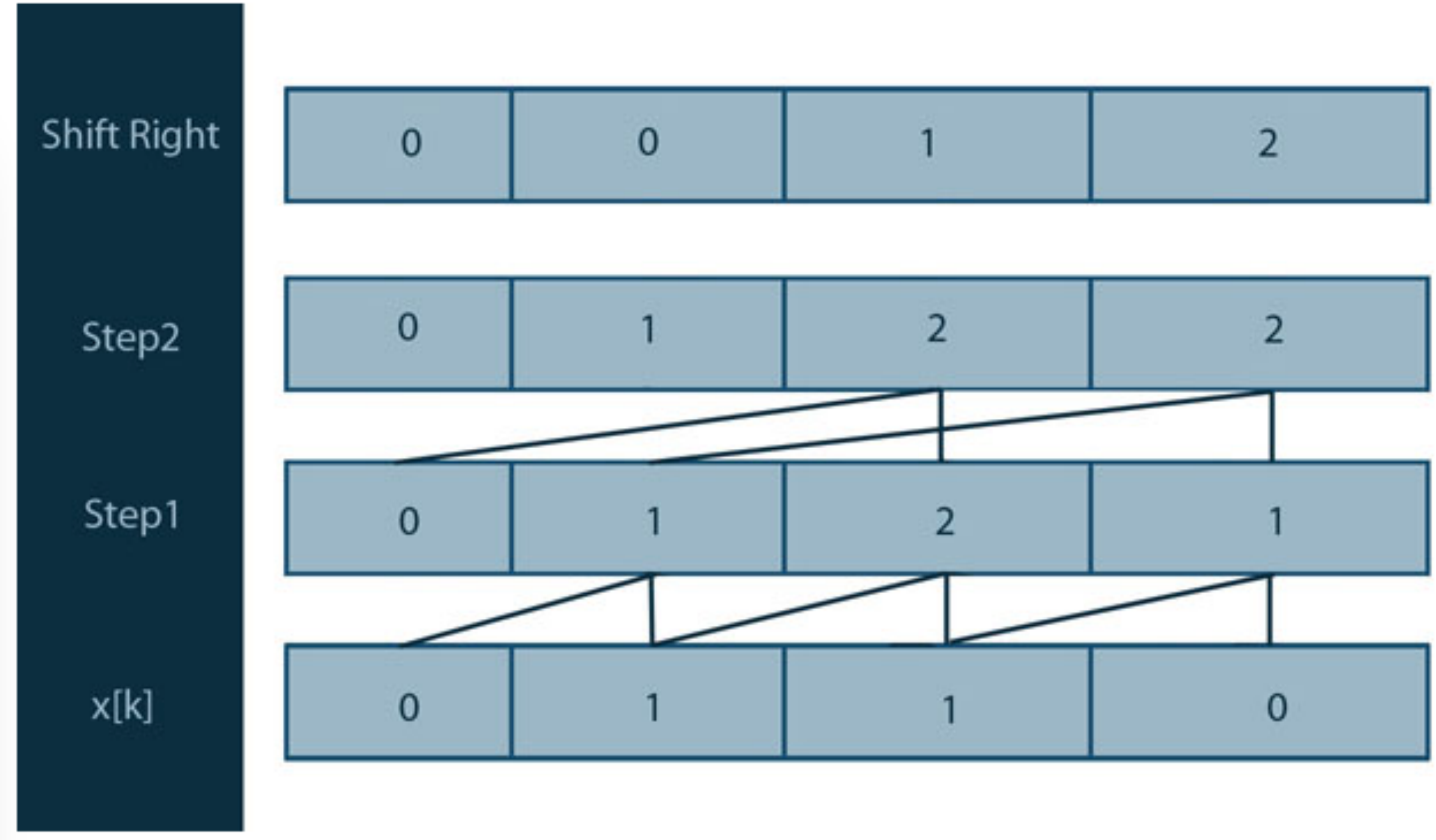
What about parallel execution?

- Computationally the sorting algorithm is reduced to computing the prefix sum arrays for each bit ranking.
- However, computing these prefix sum arrays seems to be inherently sequential. Or not?

Parallel Execution of Prefix Sums

```
Prefix_Sum (X) # X a n-bit array
  for index from 0 to log n
    DO IN PARALLEL for all k
      if k >= 2^index then
        X[k] = X[k]+X[k-2^index]
  X >> 1 #Shift all entries to the right
Return ( X )
```

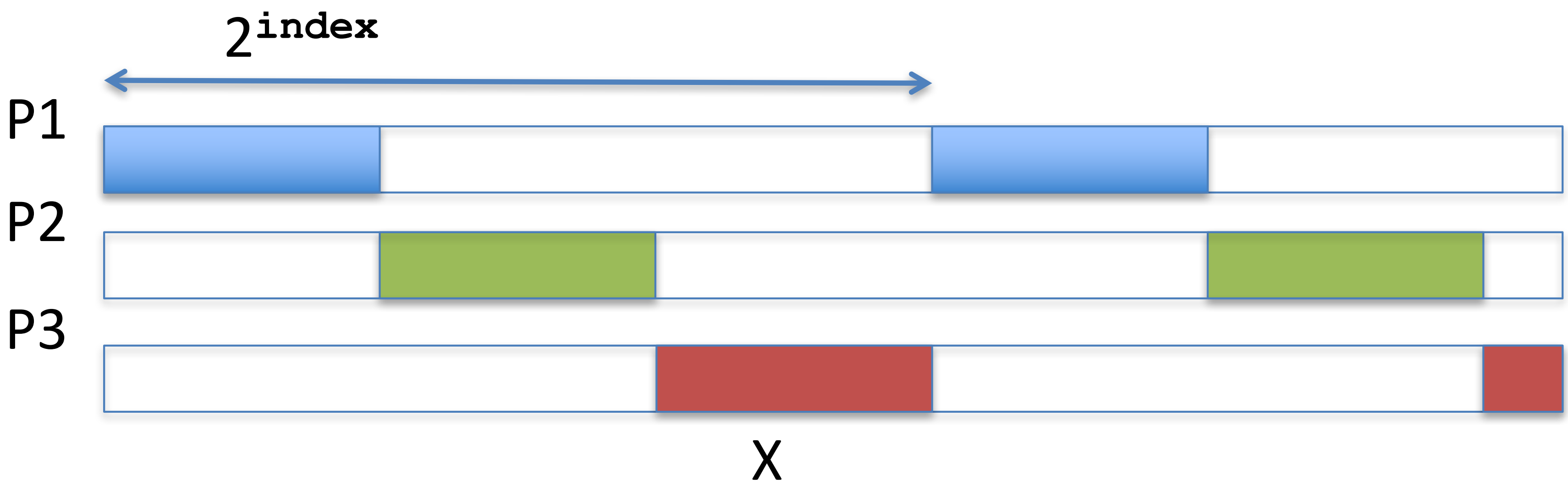
Illustration of parallel Prefix Sums



Improving Cache Performance

- The parallel prefix sum algorithm requires the whole array to be fetched at each iteration
- Bad cache performance
- Through Tiling Techniques the X array can be cut into slices (tiles)
- Once every number of iterations re-tile !!
- A CUDA implementation of the overall alg. can be found on

  https://github.com/debdattabasu/amp-radix-sort



Bitonic Sorting

Based on bitonic sequences:

A[1], A[2], …. , A[n-1], A[n] is bitonic, iff

there is a j and k such that

- A[1] … A[j] is monotonic increasing,
- A[j]…A[k] is monotonic decreasing,
- A[k]… A[n]A[1]!! is monotonic increasing

OR vise versa

A "better" definition of Bitonic Sequence

A bitonic sequence is a sequence with

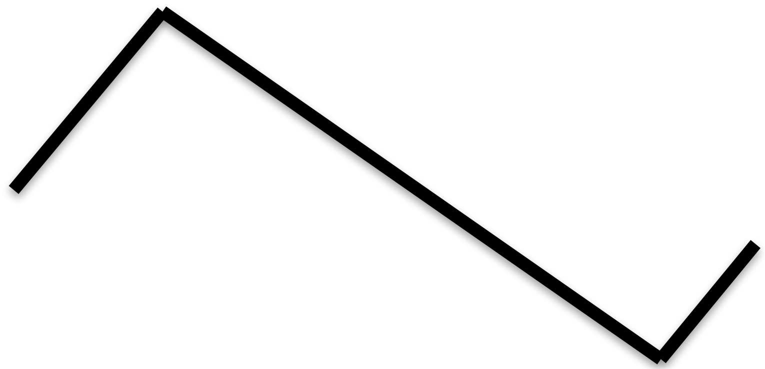
A[1]<= A[2]<= ....<=A[k]>= ... >= A[n-1]>=A[n]

for some k (1<=k<=n),

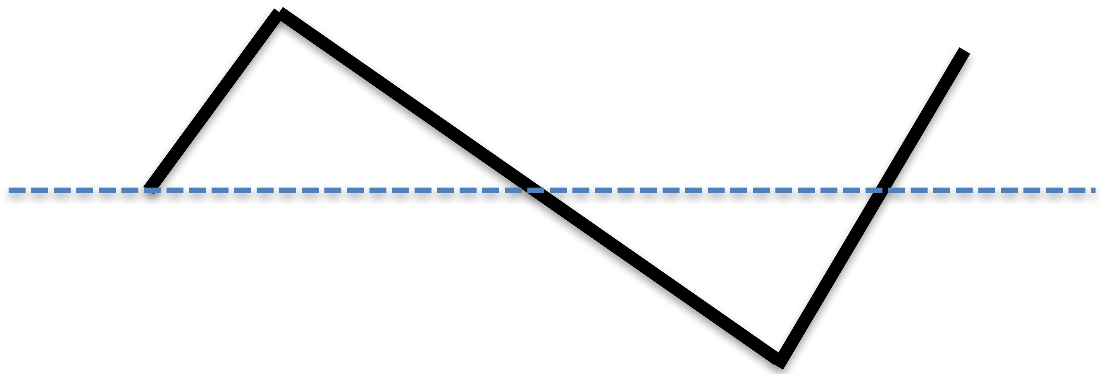
or a circular shift of such a sequence.

In a picture

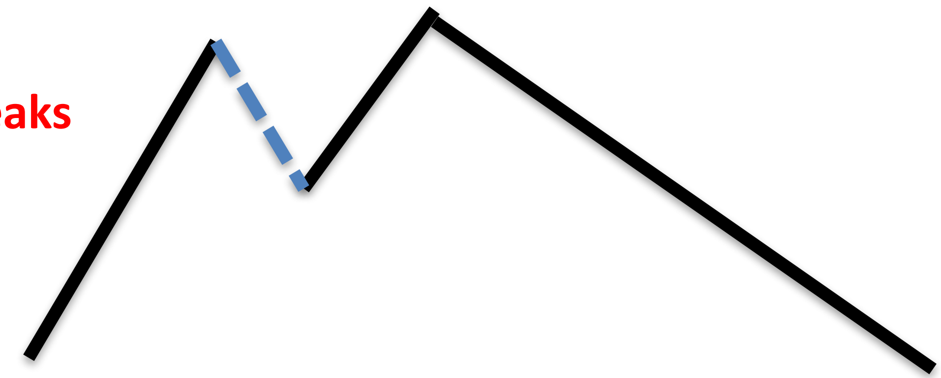
Bitonic:



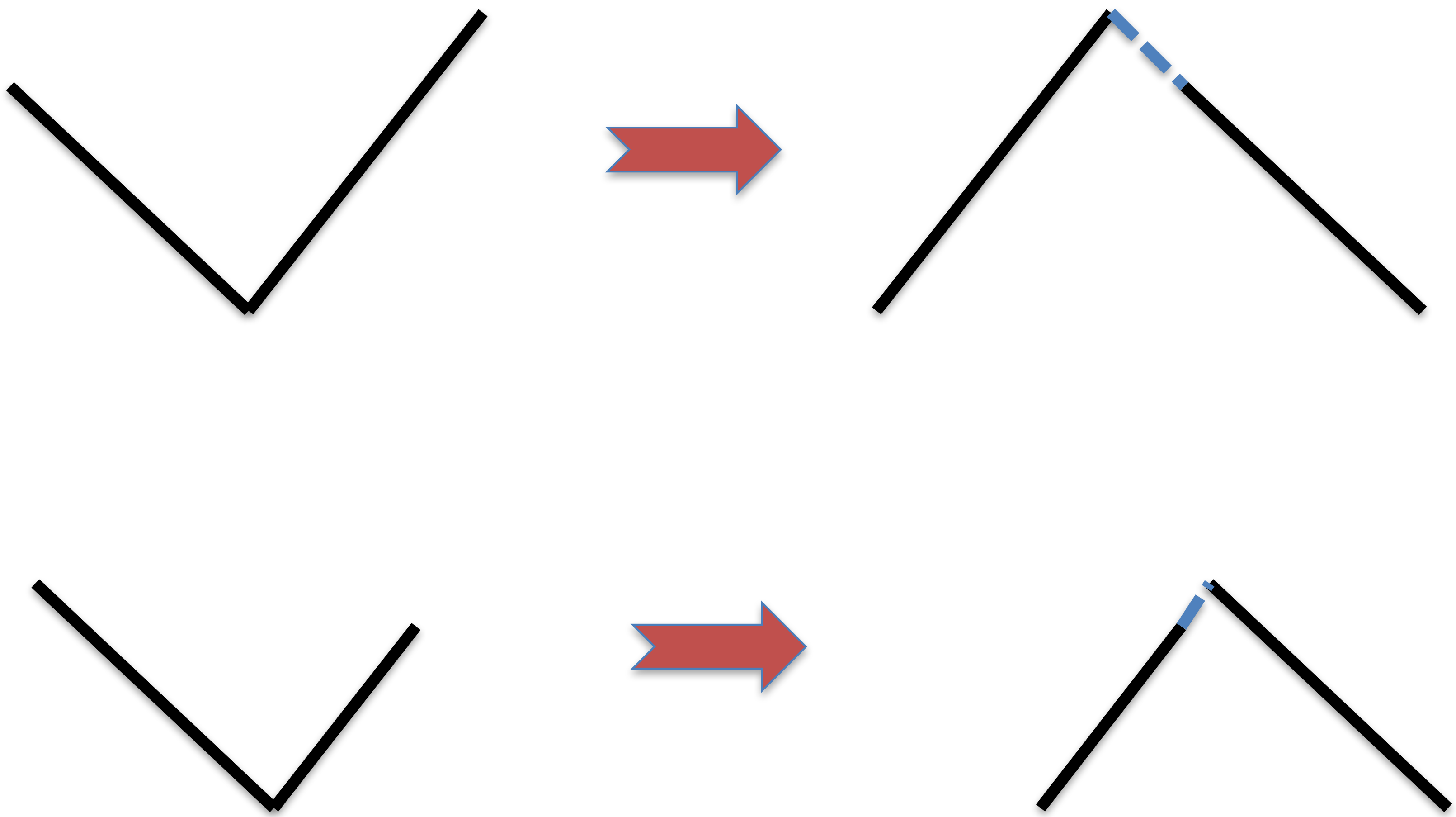
Not Bitonic



If rotated: Two Peaks



A[1]>= A[2]>= ….>=A[k]<= … <=A[n-1]<=A[n] leads to the same definition



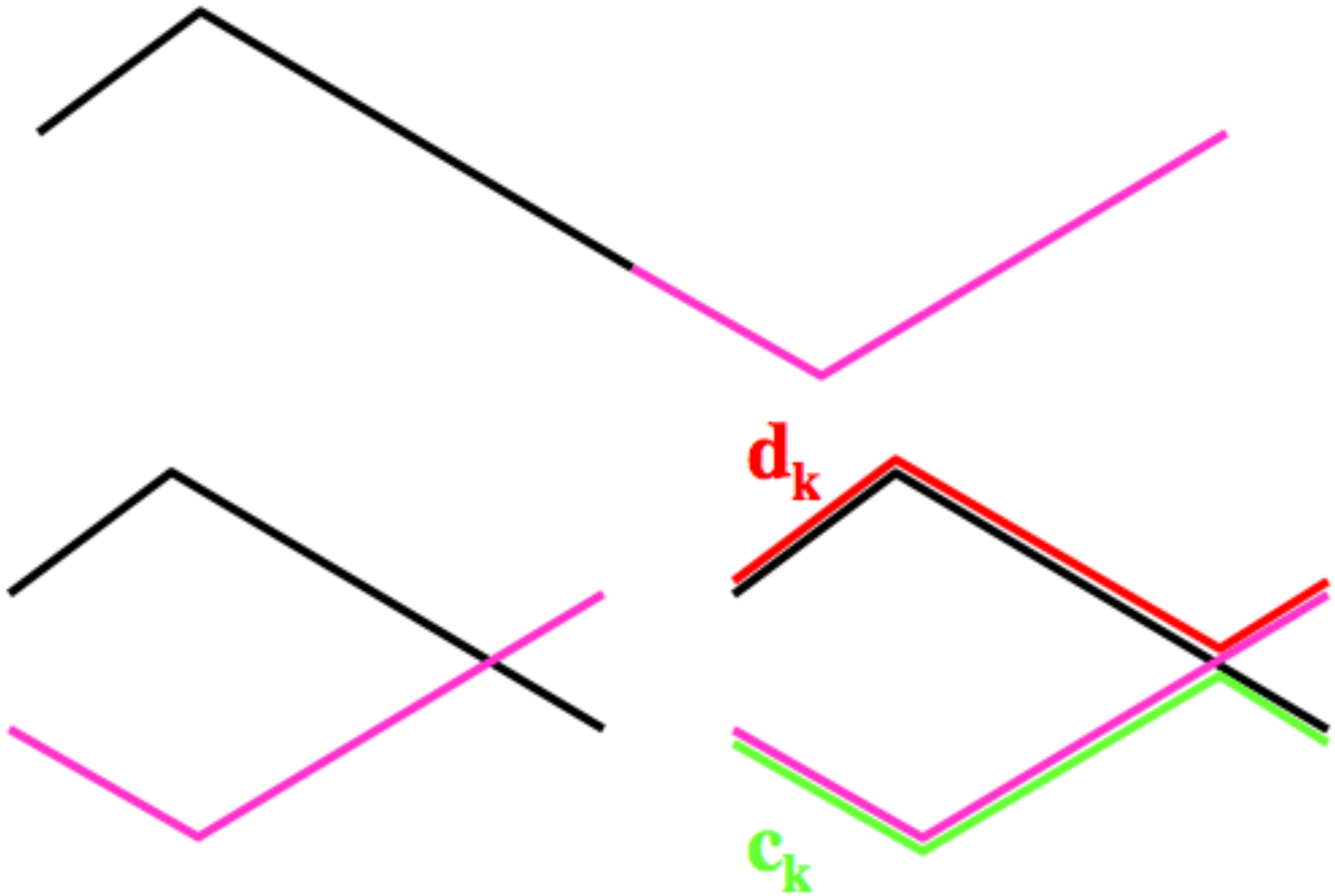
Bitonic "Merge"

```
Bitonic_Merge (A) # A is a bitonic sequence
   n = |A|
   if n == 1 then return A
   half_n = floor(n/2)
   for i from 1 to half_n
      c[i] = min(A[i],A[i+half_n])
      d[i] = max(A[i],A[i+half_n])

   DO IN PARALLEL
      Bitonic_Merge (c[1]…c[half_n])
      Bitonic_Merge (d[1]…d[half_n])

Return ( )
```

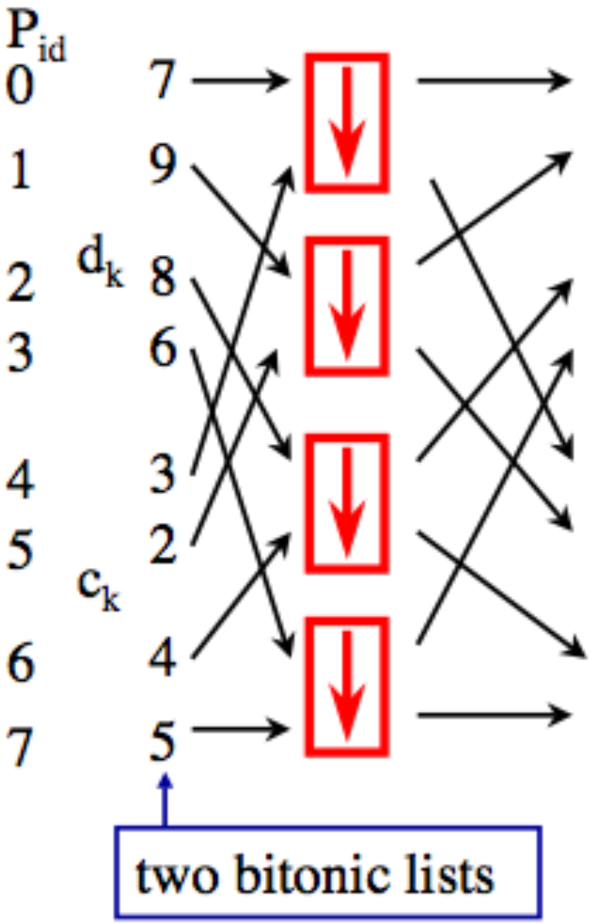
Notes Bitonic Merge

- Each `c` and `d` sequence is a bitonic sequence again
- For all `i`: `c[i] <= d[i]`
- At the end we sorted bitonic sequences of length 1, hence a sorted sequence

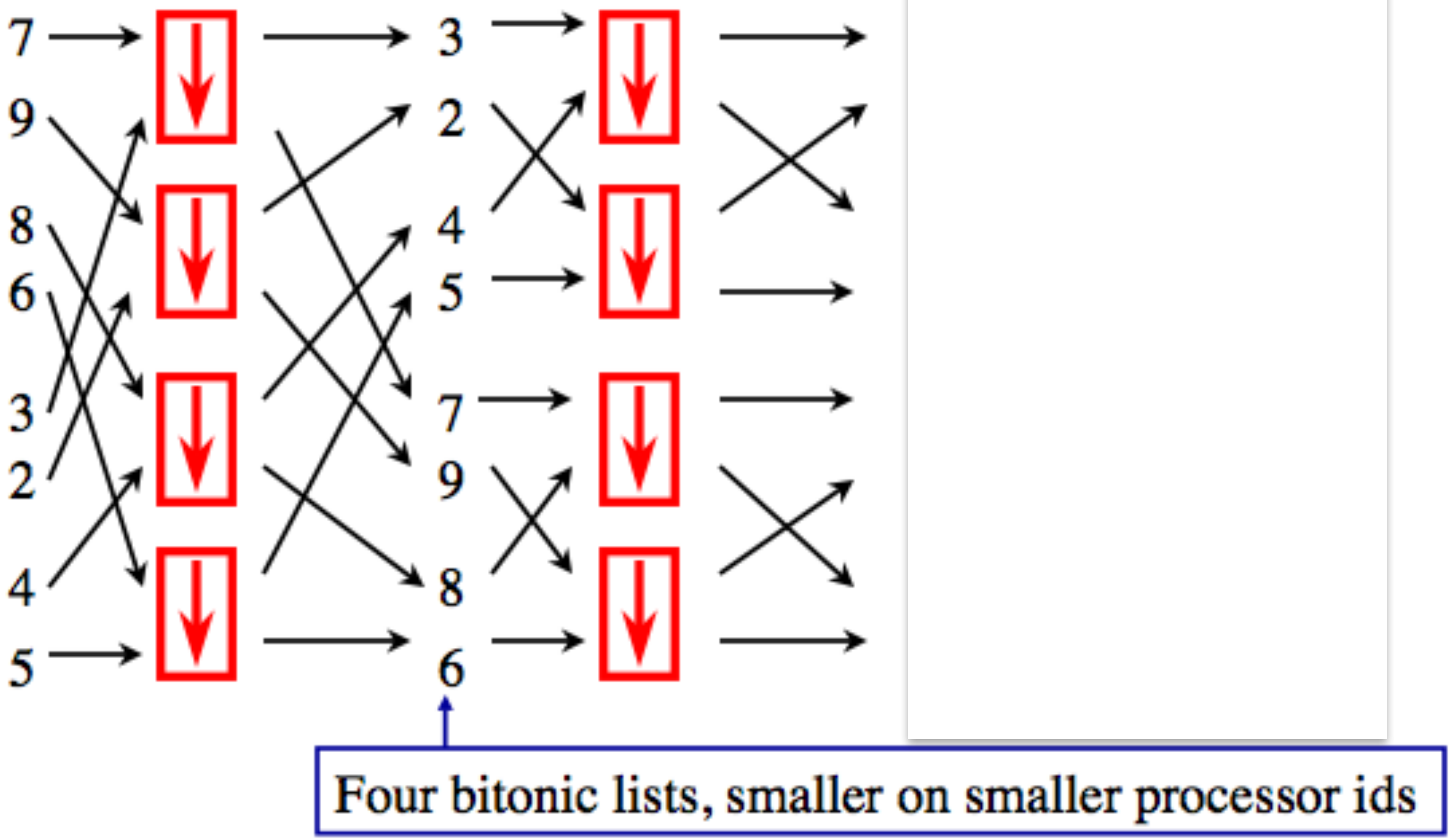
Bitonic Merge always yields bitonic sequences



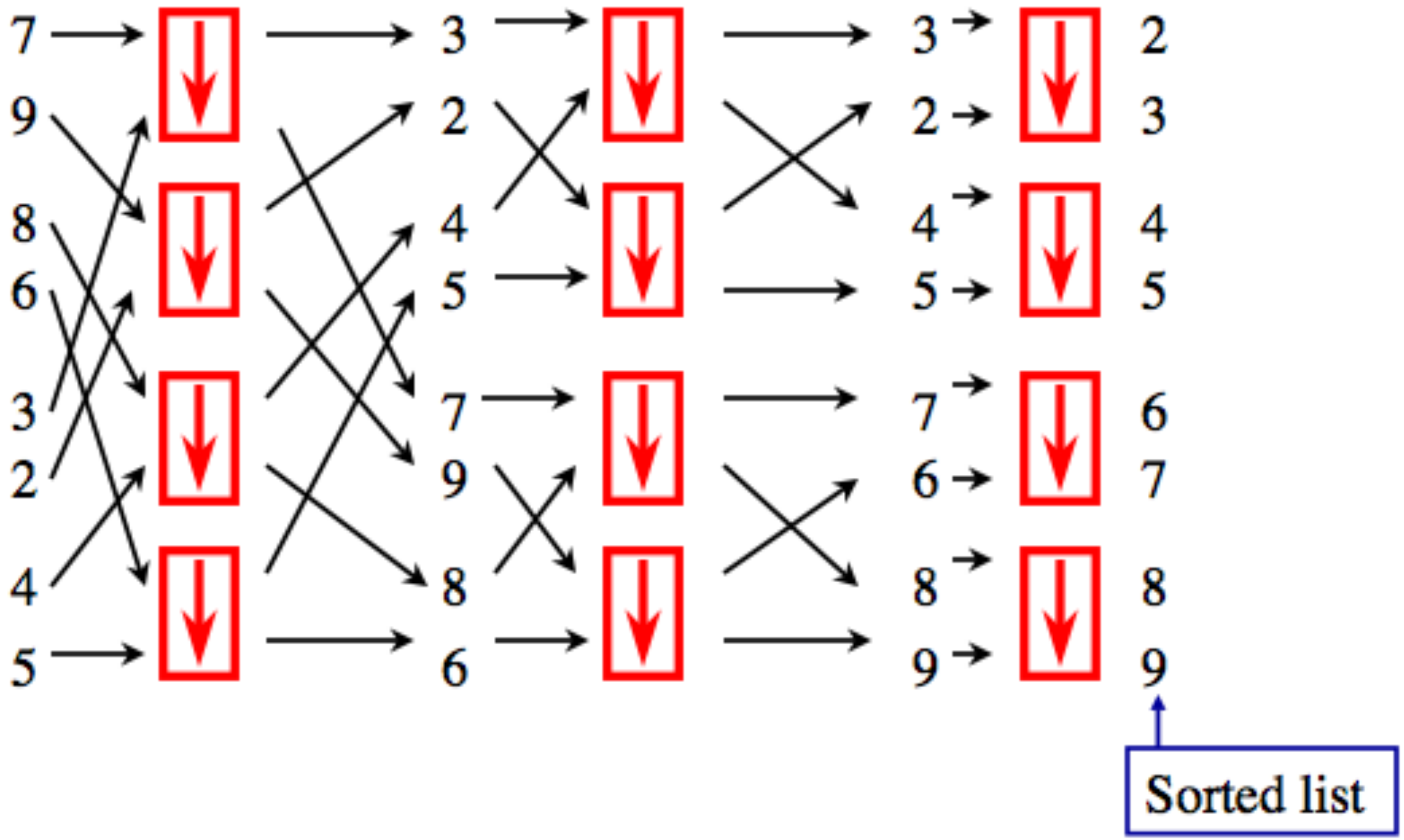
Bitonic Merge Network



Bitonic Merge Network (2)



Bitonic Merge Network (3)



Parallel Bitonic Sort

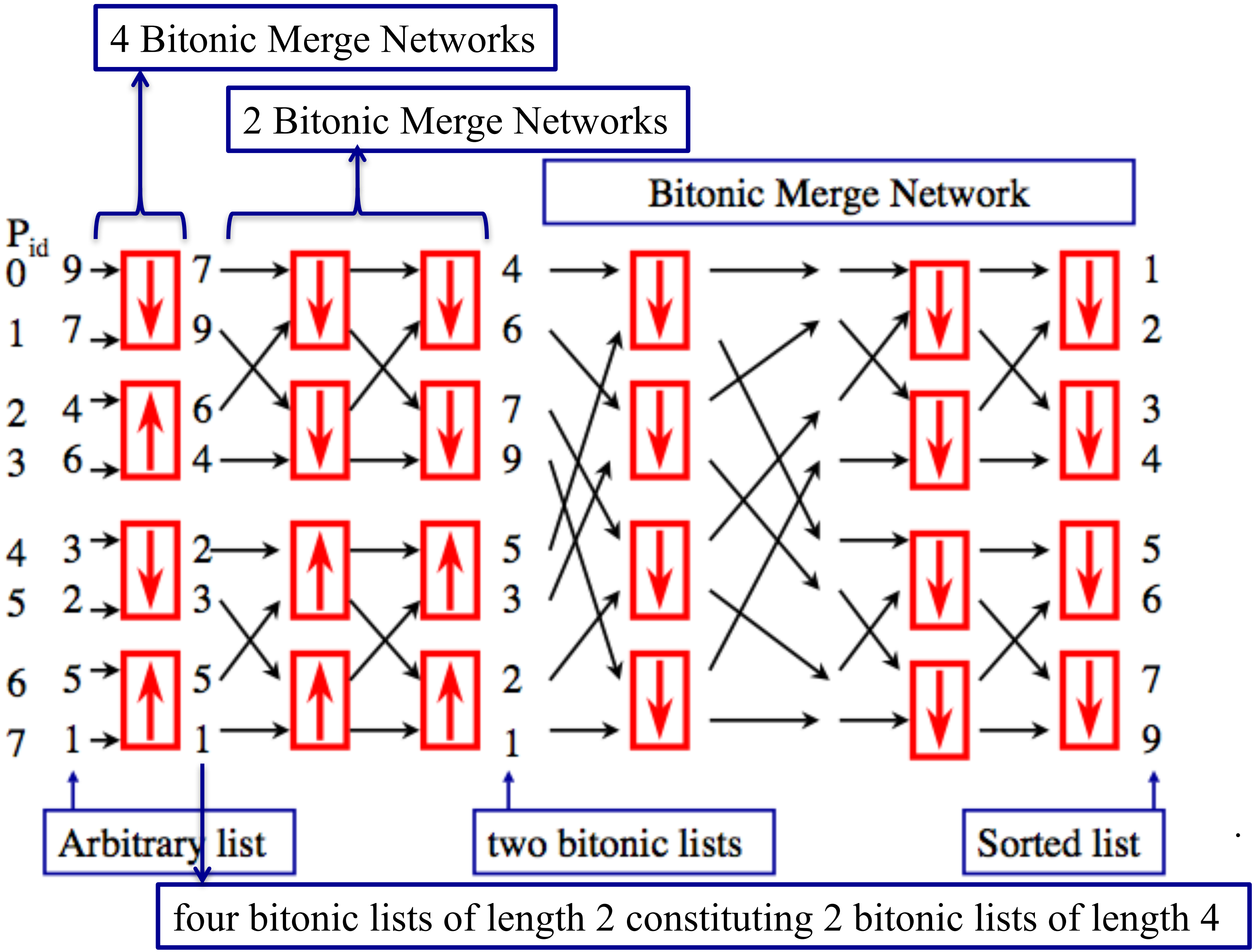
```
Bitonic_Sort (A)
   n = |A|
   if n == 1 then return A
   for i from 0 to log(n)
      DO IN PARALLEL for all k = m.2^i, k < n
         Bitonic_Merge (A[k]…A[k+2^i-1])*

Return ( )
```

*For odd values of m, interchange min and max

Notes Bitonic Sort

- Each iteration creates longer and longer bitonic sequences
- In the last iteration the whole sequence is bitonic and the final bitonic merge creates a sorted list

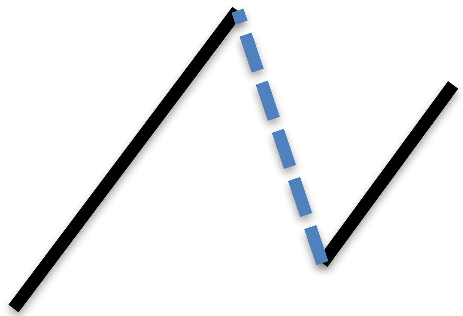
Bitonic Sort Network



Why alternating max/min?

Note that at the start of each Bitonic Merge Network we have two Bitonic Sequences which constitutes One Bitonic Sequence!!!

If one of these sequences is (monotonic) increasing and the other is (monotonic) decreasing then this is always the case. If both are increasing or decreasing this is not necessarily the case, i.e.



is not bitonic

Notes Bitonic Sort Network

- Assume n = 2^k
- The bitonic merge stages have 1, 2, 3,…,k steps each, so time to sort is

$$T(n) = 1 + 2 + \ldots + k = k\,(k-1)/2$$
$$= O\,(k^2) = O\,(\log^2 n)$$

- Each step requires n/2 processors, so the total number of processors is $O\,((n/2)\log^2 n)$
- The network can handled multiple pipelined list producing a sorted list each time step