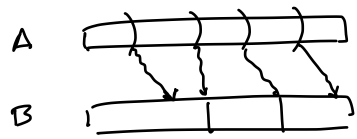
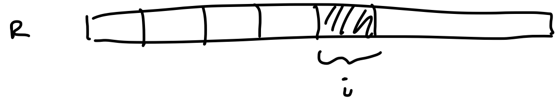
Merge (BF model)

$|A| = |B| = n$



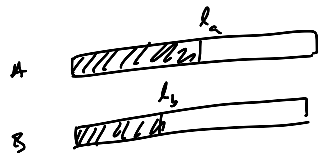
$\frac{n}{\log(n)}$

$O(\log n)$



(s_a, e_a)

(s_b, e_b)



kth (A, B, k):

return indices (l_a, l_b) s.t. $l_a + l_b = k$

and all elements in $A[0 : l_a] \cup B[0 : l_b]$

are $<$ all elts in $A[l_a : |A|] \cup B[l_b : |B|]$

- kth implementable using a dual binary search in $O(\log |A| + \log |B|)$ time (also work and depth)

Use k-th to implement merge ($f(n)$-way splitting)

MergeFWay (A, B, R) =

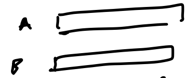
case (A, B) of

$([], \_) \Rightarrow$ copy B to R

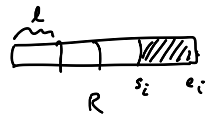
$(\_, []) \Rightarrow$ copy A to R

else $\Rightarrow$

$$l \leftarrow \frac{|R|-1}{f(|R|)} + 1 \quad \Big\} \text{ problem size}$$



$f(|R|)$ blocks, each of size l.



parfor i in $[0 : f(|R|)]$:

$\quad s \leftarrow \min(i \cdot l, |R|)$

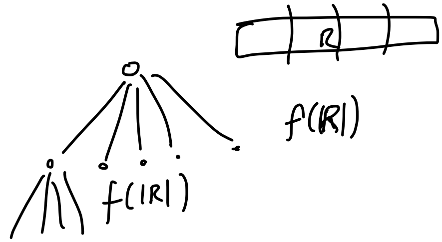
$\quad e \leftarrow \min((i+1) \cdot l, |R|)$

$\quad (s_a, s_b) \leftarrow kth(A, B, s)$

$\quad (e_a, e_b) \leftarrow kth(A, B, e)$

$\quad$ MergeFWay$(A[s_a : e_a], B[s_b : e_b], R[s : e])$

return



$f(n)$ - way divide - and - conquer

- $f(n) = \sqrt{n}$

$$W(n) = \sqrt{n}\, W(\sqrt{n}) + \sqrt{n}\log(n) \in \Theta(n)$$

$$D(n) = D(\sqrt{n}) + \underbrace{\log(n)}_{\text{parfor + k-th}}$$

$$\left.\begin{array}{l}\log(n) \\ \frac{1}{2}\log(n) \\ \frac{1}{4}\log(n) \\ \vdots\end{array}\right\} 2\log(n)$$

$2\log(n) \in O(\log n)$

$D(n) \in \Theta(\log(n))$

$W(n)$

Another choice of $f(n)$: $f(n) = \frac{n}{\log(n)}$ and combine with sequential merge

$\Rightarrow$ $O(n)$ work and $O(\log n)$ span on the (BF model).

$O(n)$ work / $O(\log n)$ depth merge

$\searrow$ $O(n \log n)$ work, $O(\log^2 n)$ depth Merge Sort (BF).

Integer-based sorting: counting sort & radix sort

- bypass the $\Omega(n \log n)$ lower bound for comparison-based sorting

Counting Sort:

Input: keys in the range [0 : m]

3 phases:

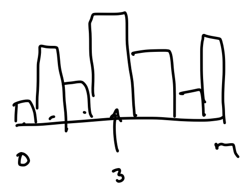
(1) count keys that have each possible value

(2) compute offsets for each key

(3) place each key in the correct position in the output.

Pseudocode:

```
CountSort(A):
    n = |A|
    m = max(A)
    counts = array(m, 0)
    offsets = array(n, 0)
    for i in [0 : |A|]:
        a_i = A[i]
        offsets[i] = counts[a_i]
        counts[a_i]++
    plusscan(counts)
    R = array(n)
    parfor i in [0 : |A|]:
        a_i = A[i]
        offset = counts[a_i] + offsets[i]
```



2 3 0 5 ...

↓

0 2 5 5 ...

↑

counts[a_i]

$R[\text{offset}] = a_i$

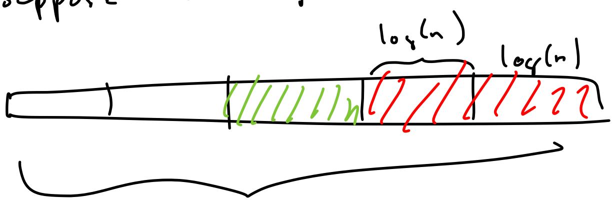
return R

$O(n+m)$ work
$O(n + \log m)$ depth } efficient if $n = m$ → $O(n)$ work

$[0 : n^c]$ $c > 1$ $O(n)$ sort?

Radix Sort

- repeatedly call counting sort.
- suppose our keys are in $[0 : n^c]$



$c \log(n)$ bits

In round i, for each key k, extract a

round key $= \frac{k}{n^i} \bmod n$

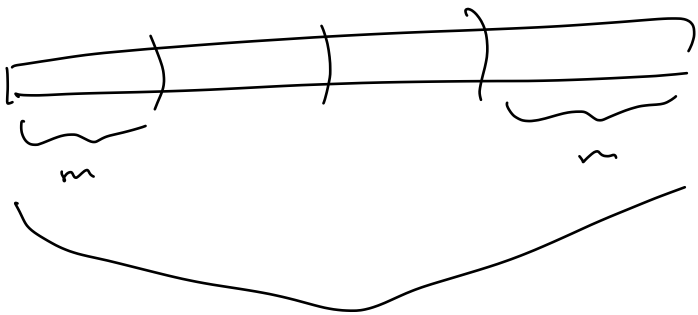
c rounds in total, $O(n)$ work each

$\Rightarrow$ $O(nc)$ work in total

<u>Open question</u> Is there a parallel integer sort that runs in $O(n)$ work and polylog span for $k \in [0, n^c]$?

<u>Thm</u>: n integers in $[0; n^c]$ in $O(\frac{n}{\alpha})$ work and $O(\frac{n^\alpha}{\alpha})$ span

Building Block 1: parallel count sort for n integers in $[0 : m]$



$b = \frac{n}{m}$ blocks

ParallelCountSort(K, V, m):

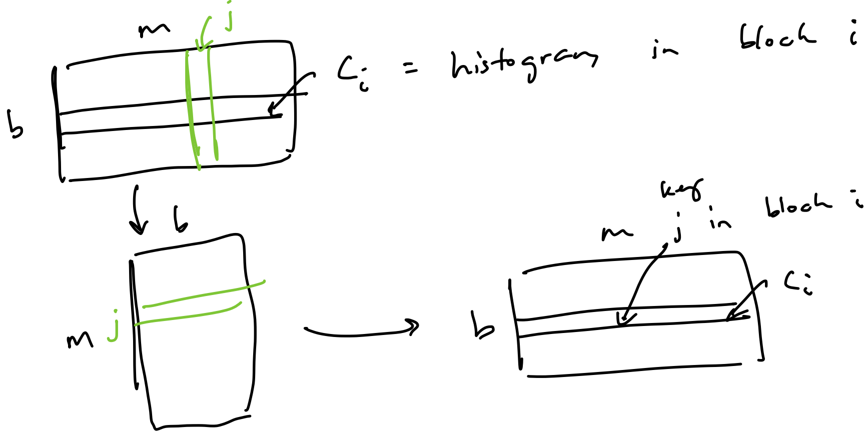
$n \leftarrow |K|$

$b \leftarrow n/m$

par for i in $[0 : b]$

$c_i \leftarrow$ counts$(K[mi : m(i+1)], m)$

sequential count sort




```
O ← plusscan(flatten(transpose(c)))

O' ← transpose(partition O into b-sized blocks)

R ← array of size |k|

parfor i in [0:b]
    place(R, O'_i, K[mi : m(i+1)], V[mi : m(i+1)], m)

return R
```

```
counts(k, m):
    for j ∈ [0 : m]:  c_j ← 0
    for j ∈ [0 : m]
        k ← k_j
        c_k ← c_k + 1
    return c
```

```
place(R, O, K, V, m):
    for j ∈ [0 : m]
        k ← k_j
        R[O_k] ← v_j
        O_k ← O_k + 1
```

Analysis:

- $\frac{n}{m}$ calls to counts / place
  - $O(m)$ work / depth each

$\Rightarrow O(n)$ work, $O(m)$ depth

- plusscan, $\underbrace{\text{flatten, transpose}}$: $O(n)$ work
  $O(\log n)$ depth

Overall: $O(n)$ work, $O(m + \log n)$ depth

To deal with $m \gg \log(n)$, "chain" together
a la sequential radix sort
- take subkeys in the range $[0 : n^{\alpha}]$, $0 < \alpha \leq 1$

$\Rightarrow$ each CS call has $O(n)$ work, $O(n^{\alpha})$ depth

$\alpha \log(n)$ bits of the input key

$\Rightarrow \frac{c}{\alpha}$ calls in total

In total: $O(\frac{n}{\alpha})$ work

$O(\frac{n^{\alpha}}{\alpha})$ depth

$\left\{ 0 : n \log^k n \right] \Rightarrow O(kn)$ work

polylog(n) depth