Parallel Sorting

CS121 Parallel Computing

Fall 2023

Outline

- Radix sort
- Merge sort
- Bitonic sort
- Sample sort

Radix sort

- Sort digit by digit, going from the least to most significant digit.
- Sort must be stable. If there’s tie on current digit, must preserve order from previous digits.
  - Ex When sorting 100s digit, there’s a tie on value 3. Preserve earlier order, i.e. 362 before 397.
- Sorting each digit (or group of digits) highly parallel.
- Radix sort is typically one of the fastest sorts in practice.

362	291	207	207
436	362	436	253
291	253	253	291
487	436	362	362
207	487	487	397
253	207	291	436
397	397	397	487

Radix sort and prefix sum

- We'll sort the last digits of a set of binary numbers in a stable way.
  - Call elements ending in 0 0-vals, the rest1-vals.
- Goal is to put the 0-vals before the 1-vals in a stable way.
  - 0-val at index i goes to (# 0-vals before i).
  - 1-val at index i goes to (total # 0-vals) + (# 1-vals before i) = (total # 0-vals) + (i - # 0-vals before i).
- Use prefix sum to count # 0-vals up to every index.

100	111	010	110	011	101	001	000	Input Array
0	1	0	0	1	1	1	0	least significant bit
1	0	1	1	0	0	0	1	e = flip the bits
0	1	1	2	3	3	3	3	f = prefix sum

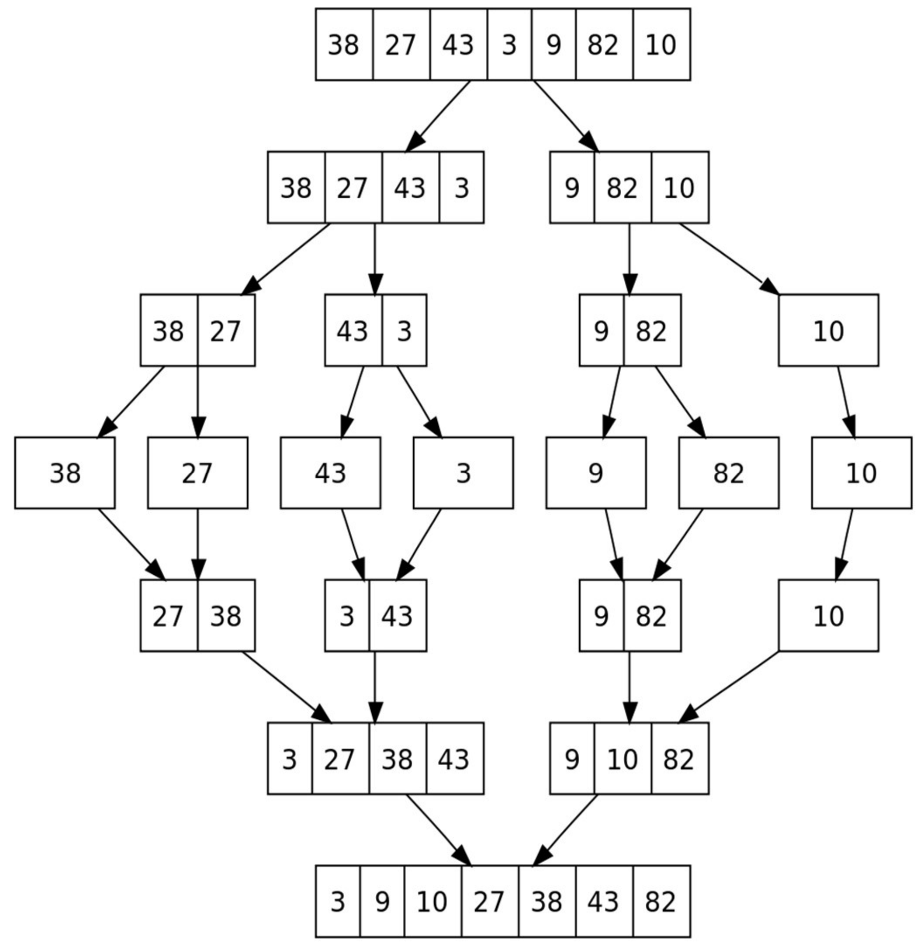
Total # 0's = e[n-1] + f[n-1]

0-0+4 = 4	1-1+4 = 4	2-1+4 = 5	3-2+4 = 5	4-3+4 = 5	5-3+4 = 6	6-3+4 = 7	7-3+4 = 8	t = index – f + total # 0's
0	4	1	2	5	6	7	3	d = b ? t : f
100	111	010	110	011	101	001	000	Scatter input using d as scatter address
100	010	110	000	111	011	101	001	

http://www.seas.upenn.edu/~cis565/LECTURE2010/CUDALibariesandTools.ppt

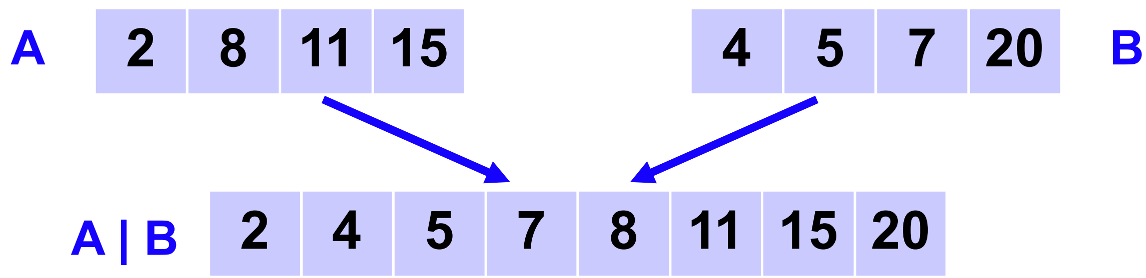
Parallel mergesort

- Divide and conquer sort in which subproblems can be solved in parallel.
- There are log n divide stages, followed by log n merge stages.
- Each merge stage takes O(n) sequential time.
- We'll do each merge stage in O(log n) parallel time with n processors.
- So $O(\log^2 n)$ time to sort n numbers with n processors.
- Assume for simplicity all values are unique.



https://en.wikipedia.org/wiki/Merge_sort

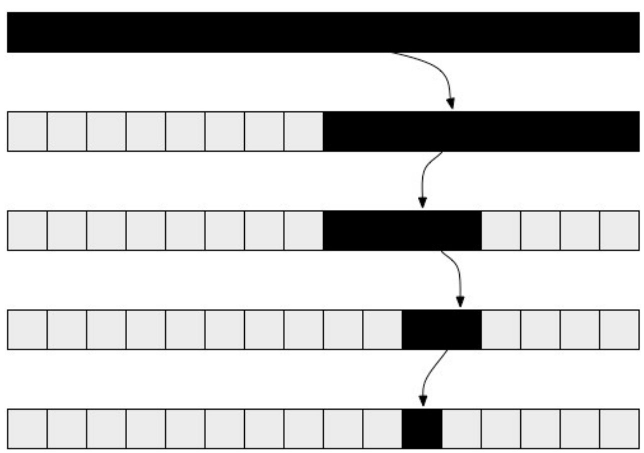
Parallel merge



- rank(x,S) = |{y≤x | y∈S}| = number of values in S less than or equal to x.
  - Ex rank(8,A)=2, rank(8,B)=3, rank(20,A)=4.
- Claim Let x∈A∪B, then rank(x, A | B) = rank(x,A) + rank(x,B).
  - Ex rank(8, A | B) = 5 = rank(8,A)+rank(8,B) = 2+3.
  - Ex rank(20, A | B) = 8 = rank(20,A)+rank(20,B) = 4+4.
- Proof Say x∈A.
  - There are rank(x,A) elements ≤ x in A, including x itself, and rank(x,B) elements ≤ x in B, so a total of rank(x,A)+rank(x,B) elements ≤ x in A∪B.

Parallel merge

- If S is sorted array of size n, can compute rank(x,S) in O(log n) sequential time.
  - Do binary search for x in S.
  - Say search ends at index i. If S[i]=x, return i+1, else return i.
  - Ex x=11, S=[4,5,7,20], search ends at index 3, so rank(x,S)=3.



Parallel merge

- Let A, B be sorted arrays with n elements each.
- We compute A | B using 2n processor in O(log n) time.
- Output stored in array C of size 2n.

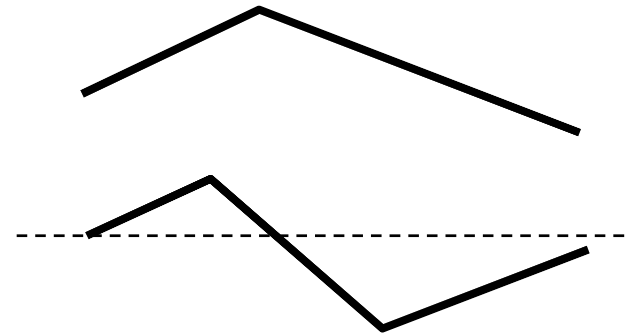
- For $1 \le i \le n$, processor i computes r_i=rank(A[i],B).
  - Write A[i] to $C(i+r_i)$.
- For $1 \le j \le n$, proc j+n computes r_j=rank(B[j],A).
  - Write B[j] to $C(j+r_j)$.

rank(A[i],B)	0	3	3	3
A	**2**	**8**	**11**	**15**

B	**4**	**5**	**7**	**20**
rank(B[j],A)	1	1	1	4

A | B	**2**	**4**	**5**	**7**	**8**	**11**	**15**	**20**
index	1	2	3	4	5	6	7	8

Bitonic sort

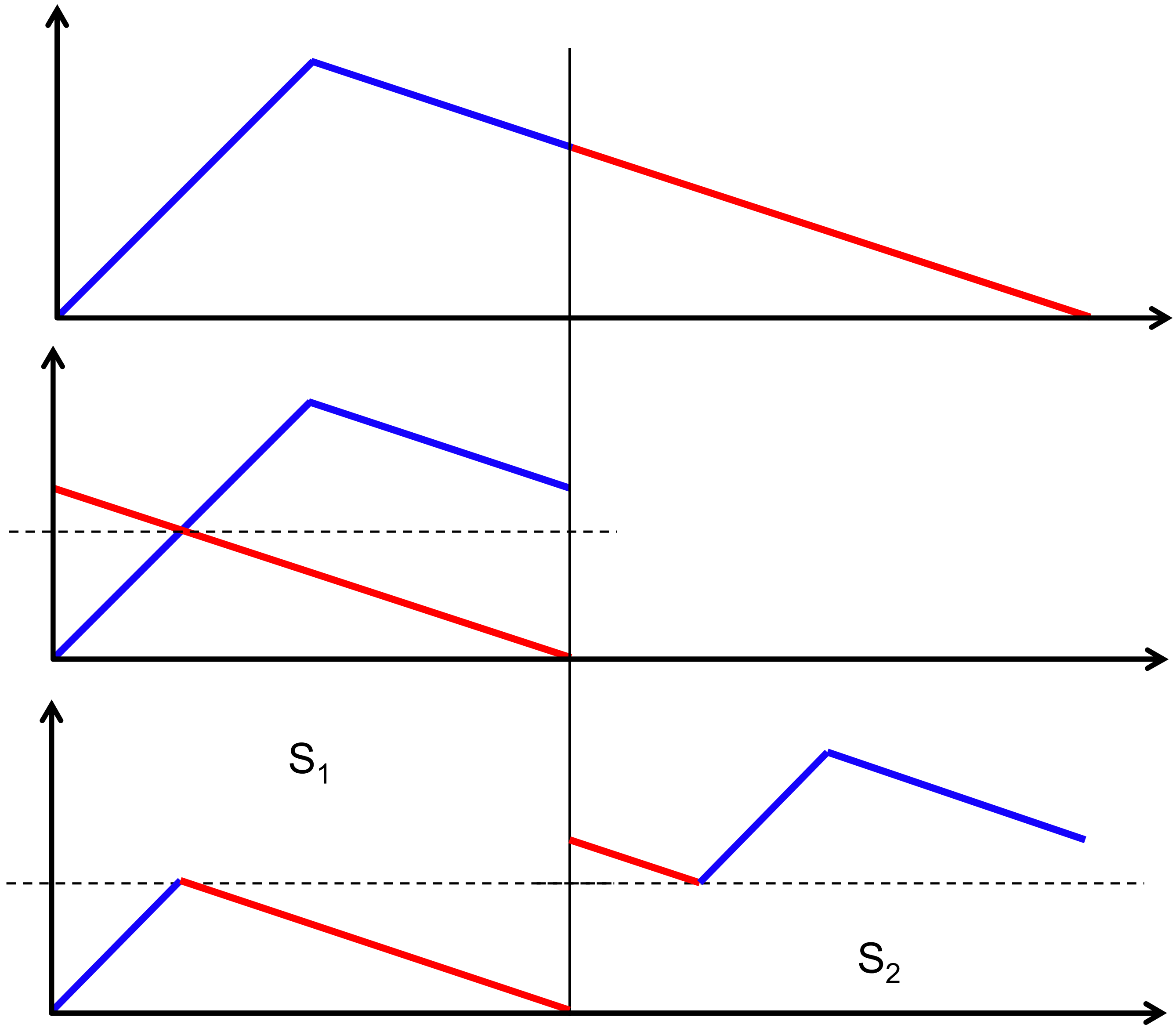


- A bitonic sequence is one that
  - First increases, then decreases.
  - Or is the rotation of a sequence of the first kind.
- Ex [1,3,4,7,8,5,2,1,0] is a bitonic sequence
- Ex [7,8,5,2,1,0,1,3,4] is a bitonic sequence, because it's a rotation of the first example.
- Lemma Let $[a_0,a_1,\ldots,a_{n-1}]$ be a bitonic sequence, and let

$$S_1 = [\min(a_0, a_{n/2}), \min(a_1, a_{n/2+1}), \ldots, \min(a_{n/2-1}, a_{n-1})]$$
$$S_2 = [\max(a_0, a_{n/2}), \max(a_1, a_{n/2+1}), \ldots, \max(a_{n/2-1}, a_{n-1})]$$

  Then S_1 and S_2 are both bitonic sequences, and all elements of S_1 are $\leq$ all elements of S_2.
- This operation is called bitonic split.

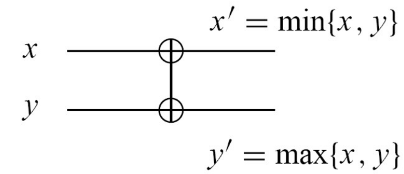
Proof of lemma

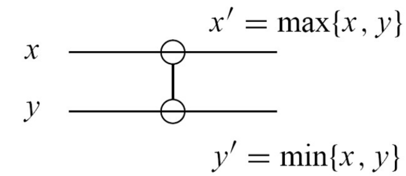


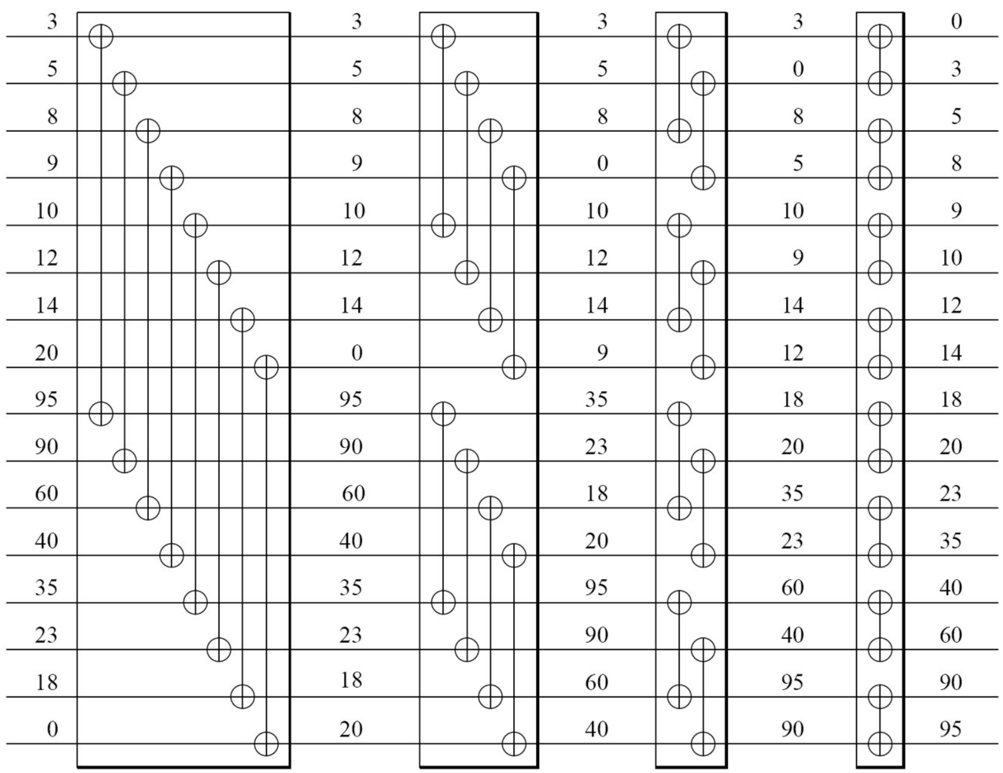
Bitonic merge

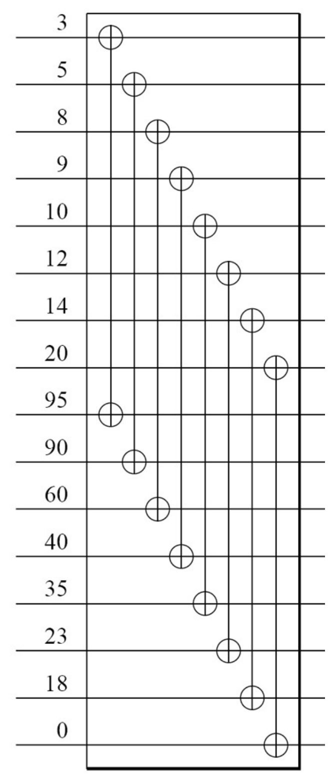
- Bitonic merge takes a bitonic sequence and converts it to a sorted one using a sequence of bitonic splits.
- Given a bitonic sequence S, a bitonic split "sorts" S in the sense that the first half of S is $\leq$ the second half of S after the split.
- Now we can split each half recursively, to sort more finely, into quarters.
- Finally, after we split down to sequences of size 1, the entire sequence is sorted in nondecreasing order.

3	5	8	9	10	12	14	20	95	90	60	40	35	23	18	0
3	5	8	9	10	12	14	0	95	90	60	40	35	23	18	20
3	5	8	0	10	12	14	9	35	23	18	20	95	90	60	40
3	0	8	5	10	9	14	12	18	20	35	23	60	40	95	90
0	3	5	8	9	10	12	14	18	20	23	35	40	60	90	95


$x' = \min\{x, y\}$
x
y
$y' = \max\{x, y\}$


$x' = \max\{x, y\}$
x
y
$y' = \min\{x, y\}$

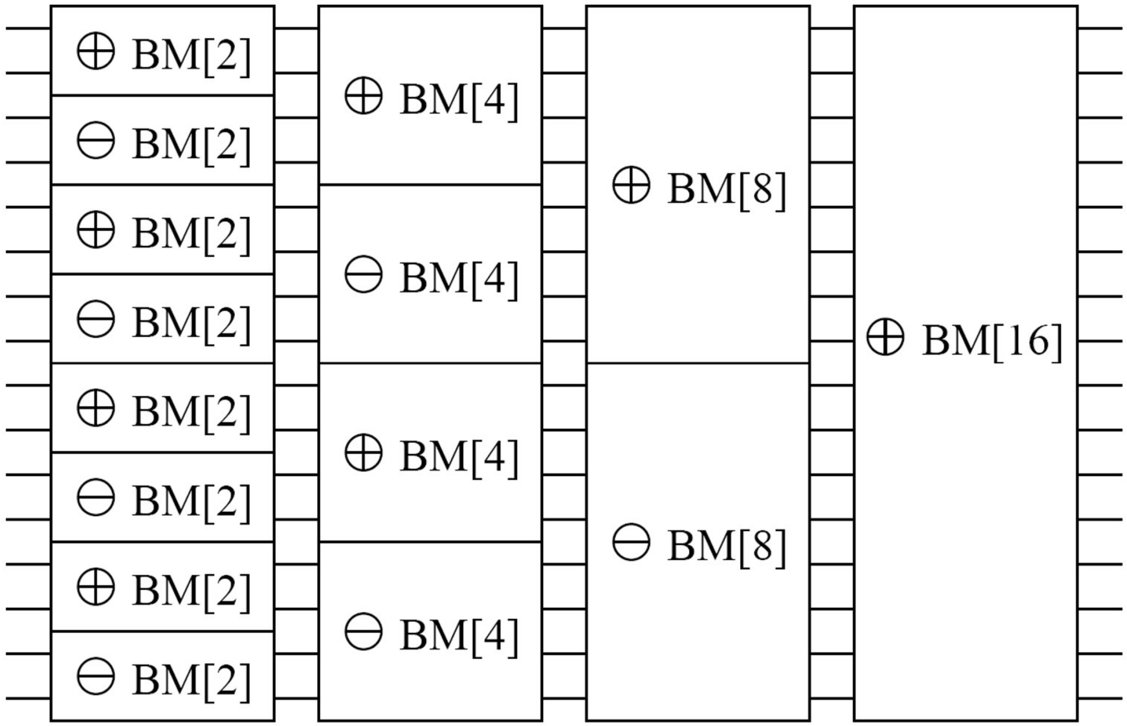

3 5 8 9 10 12 14 20 95 90 60 40 35 23 18 0
3 5 8 9 10 12 14 0 95 90 60 40 35 23 18 20
3 5 8 0 10 12 14 9 35 23 18 20 95 90 60 40
3 0 8 5 10 9 14 12 18 20 35 23 60 40 95 90
0 3 5 8 9 10 12 14 18 20 23 35 40 60 90 95


3 5 8 9 10 12 14 20 95 90 60 40 35 23 18 0

Bitonic sort

- Bitonic merge can produce either an increasing or decreasing sequence.
  - Call these BM$\oplus$ and BM$\ominus$.
- To sort an arbitrary size n sequence
  - First, convert it to a bitonic sequence, with each part of size n/2.
  - Do bitonic merge on the sequences.
- To convert the sequence to a bitonic one
  - Divide the sequence in half.
  - Sort the first half in increasing order.
  - Sort the second half in decreasing order.
  - Each sort is done recursively.
  - When we reach sequence of size 2, it's automatically bitonic.

Bitonic sort network



- There are log n bitonic merges.
- Each bitonic merge takes ≤ log n time.
- Bitonic merge takes O($\log^2$ n) parallel time total.
- Not work efficient, since total work is O(n $\log^2$ n).
- Work efficient sorting networks exist, e.g. the AKS network, but have high constant factors and aren't practical.

Sample sort

- Sample sort is often used in distributed memory setting.
- Given p processors to sort n numbers, ideally each processor sorts n/p numbers.
- To do this, pick p-1 pivots, say $t_1 < t_2 < \ldots < t_{p-1}$. Let $t_0 = m$ and $t_p = M$, where m and M are min and max inputs.
  - Form p buckets, where i'th bucket contains all inputs between t_{i-1} and t_i.
  - i'th processor sorts i'th bucket sort locally.
  - If S is the max bucket size, sorting takes O(S log S) parallel time.
- Main problem with this approach is buckets are unlikely to be balanced.
  - For example, if we pick pivots randomly, it's likely $S = \Theta(n \log n / p)$, so sorting takes $\Theta(n \log^2 n / p)$ instead of the optimal $\Theta(n \log n / p)$.

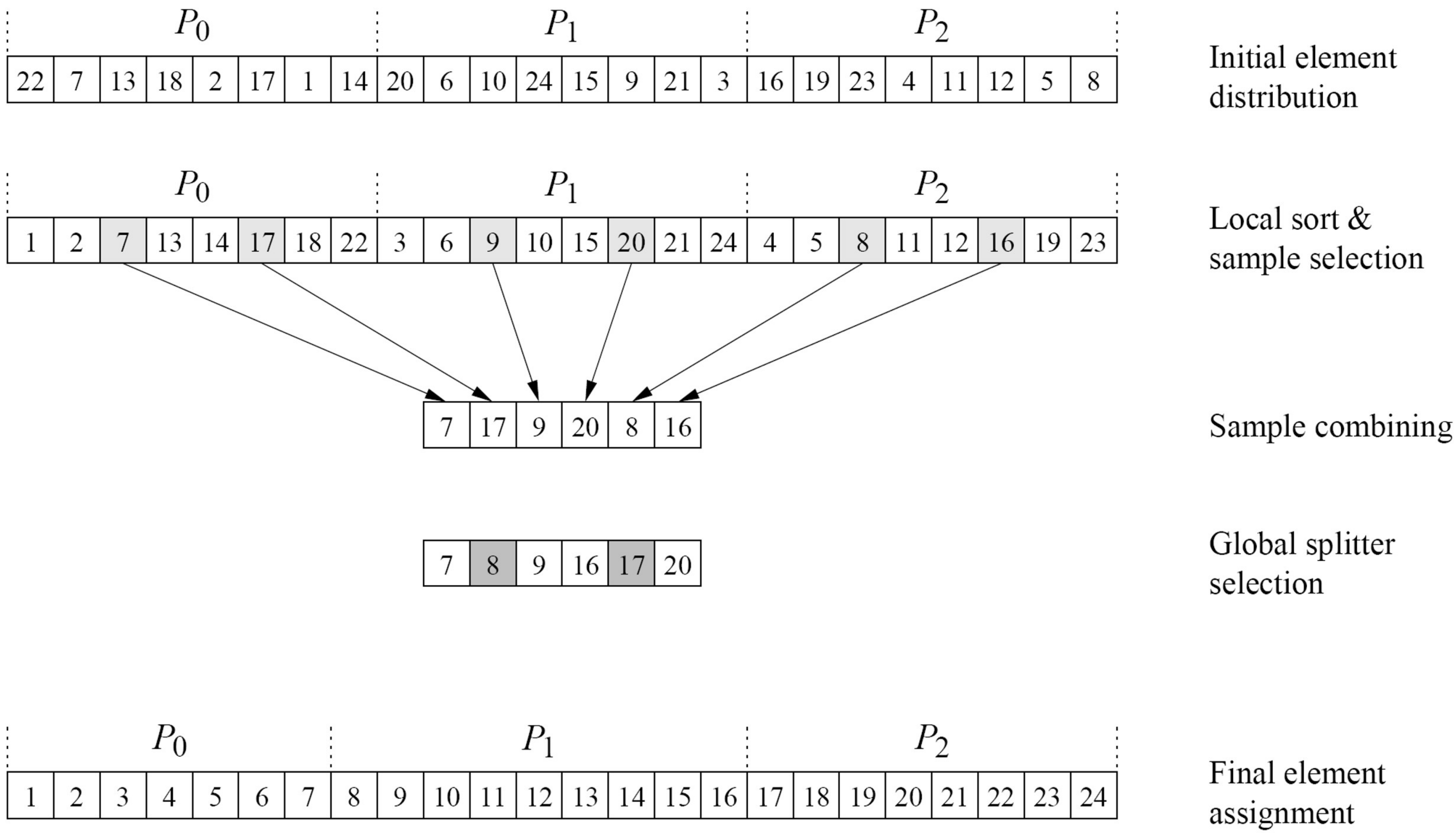
Sample sort

- Sample sort evens out the bucket sizes, so $S = \Theta(n / p)$ with high probability.
  - Sample $r = \lambda p$ random elements, for $\lambda > 1$ given later.
  - Sort the sampled elements and pick every λ'th sample as a pivot, producing p pivots.
  - Use the pivots to form buckets, as earlier.
- Thm If $\lambda = 12 \ln(n)$, then no bucket is larger than $4n/p$ with probability at least $1-1/n^2$.
  - Proof based on Chernoff bound, which bounds probability a sum of independent random variables deviates substantially from its expectation.
- Sample sort runs in $\Theta(n \log n / p)$ with high probability.
- It also has low communication complexity, since it only needs to broadcast the pivots and communicate to form the buckets.

Sample sort algorithm

- Each processor starts with n/p values.
- Each processor picks λ random values and sends them to processor 1.
- Processor 1 sorts λp values sequentially.
  - Choose set S with every λ'th value as pivots.
- Processor 1 broadcasts S to all other processors.
- Each processor uses S to form p buckets for its values.
- Each processor sends values from the i'th bucket to the i'th processor.
- Each processor sorts the values it receives sequentially.

Example



P_0 P_1 P_2
22 7 13 18 2 17 1 14 20 6 10 24 15 9 21 3 16 19 23 4 11 12 5 8
Initial element distribution
P_0 P_1 P_2
1 2 7 13 14 17 18 22 3 6 9 10 15 20 21 24 4 5 8 11 12 16 19 23
Local sort & sample selection
7 17 9 20 8 16
Sample combining
7 8 9 16 17 20
Global splitter selection
P_0 P_1 P_2
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
Final element assignment