CME 213, ME 339—Spring 2021

Eric Darve, ICME, Stanford

Stanford University

“Optimism is an occupational hazard of programming; feedback is the treatment.” (Kent Beck)

Sorting algorithms on shared memory computers

Homework 2 focuses on **radix sort**

Applies to integers or floats only

Uses buckets

Partitions the bits into small groups

Order using groups of bits and buckets

Radix sort animations

Musical demo MSD

Musical demo LSD

Most/least significant digit

Quicksort

One of the fastest sorting algorithms

Quicksort algorithm

Divide and conquer approach. Divide step:

- Choose a pivot x
- Separate sequence into 2 sub-sequences with all elements smaller than x and greater than x

Conquer step:

- Sort the two subsequences

```
Gray = currently not being sorted; blue = pivot; red = swap
[  3  156  56  205  93  159  180  209  166  223  198  164  231  244  251  126  ]
[  3  156  56  205  93  159  180  209  166  223  198  164  231  244  251  126  ]
[  3  156  56  205  93  159  180  209  166  223  198  164  231  244  251  126  ]
[  3  156  56   93 205  159  180  209  166  223  198  164  231  244  251  126  ]
[  3  156  56   93 126  159  180  209  166  223  198  164  231  244  251  205  ]
[  3  126  56   93 156  159  180  209  166  223  198  164  231  244  251  205  ]
[  3  126  56   93 156  159  180  209  166  223  198  164  231  244  251  205  ]
[  3  126  56   93 156  159  180  209  166  223  198  164  231  244  251  205  ]
[  3  93  56  126  156  159  180  209  166  223  198  164  231  244  251  205  ]
[  3  93  56  126  156  159  180  209  166  223  198  164  231  244  251  205  ]
[  3  56  93  126  156  159  180  209  166  223  198  164  231  244  251  205  ]
[  3  56  93  126  156  159  180  209  166  223  198  164  231  244  251  205  ]
[  3  56  93  126  156  159  180  166  209  223  198  164  231  244  251  205  ]
[  3  56  93  126  156  159  180  166  164  223  198  209  231  244  251  205  ]
[  3  56  93  126  156  159  164  166  180  223  198  209  231  244  251  205  ]
[  3  56  93  126  156  159  164  166  180  223  198  209  231  244  251  205  ]
[  3  56  93  126  156  159  164  166  180  223  198  209  231  244  251  205  ]
[  3  56  93  126  156  159  164  166  180  223  198  209  231  244  251  205  ]
[  3  56  93  126  156  159  164  166  180  223  198  209  205  244  251  231  ]
[  3  56  93  126  156  159  164  166  180  205  198  209  223  244  251  231  ]
[  3  56  93  126  156  159  164  166  180  205  198  209  223  244  251  231  ]
[  3  56  93  126  156  159  164  166  180  198  205  209  223  244  251  231  ]
[  3  56  93  126  156  159  164  166  180  198  205  209  223  244  231  251  ]
[  3  56  93  126  156  159  164  166  180  198  205  209  223  231  244  251  ]
```

```python
def quicksort(A,l,u):
    if l < u-1:
        x = A[l]
        s = l
        for i in range(l+1,u):
            if A[i] <= x: # Swap entries smaller than pivot
                s = s+1
                A[s], A[i] = A[i], A[s]
        A[s], A[l] = A[l], A[s]
        quicksort(A,l,s)
        quicksort(A,s+1,u)
```

Python code

On average, it runs very fast, even faster than mergesort.

It requires no additional memory

[Musical demo LL pointers]

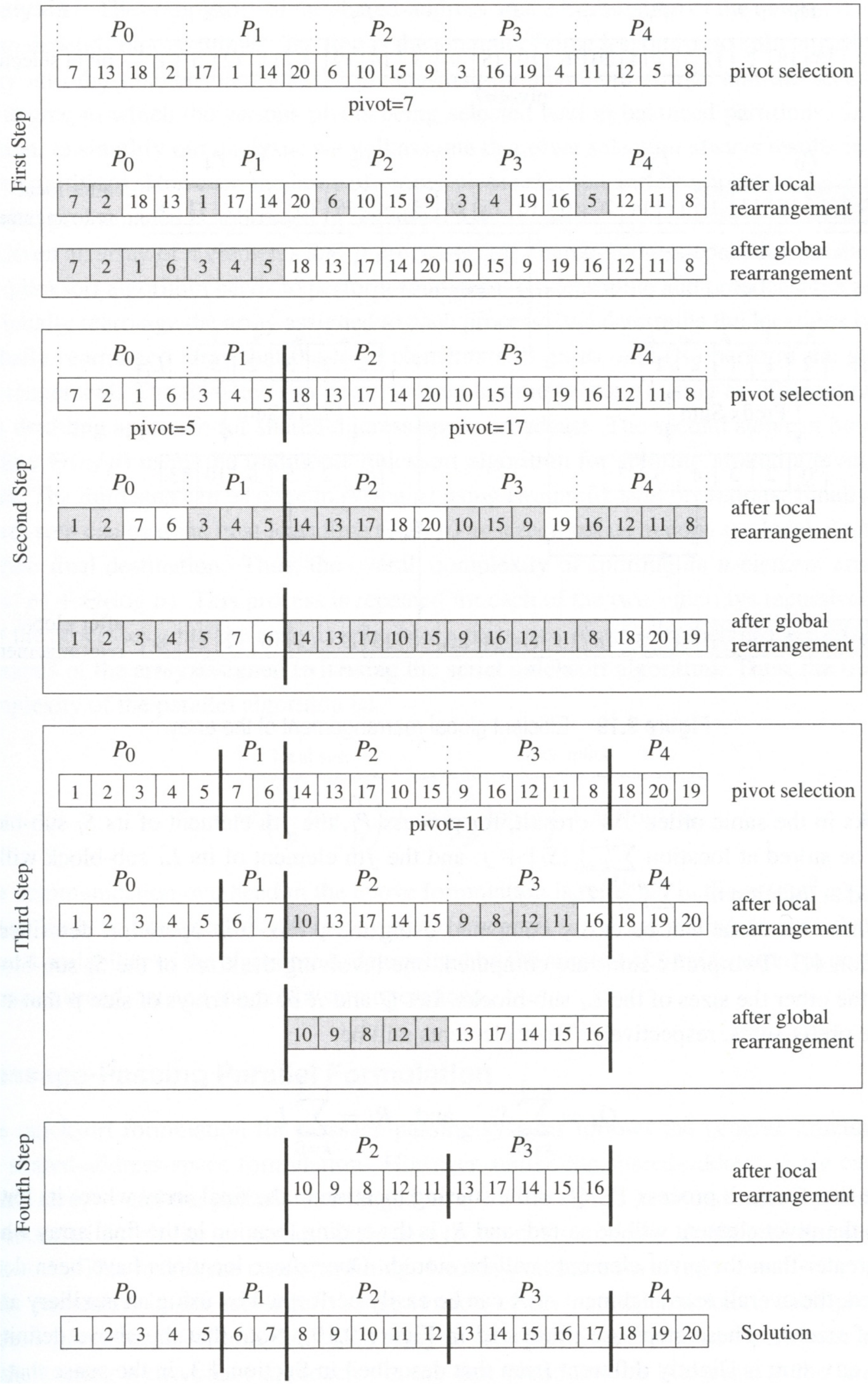
[Musical demo LR pointers]

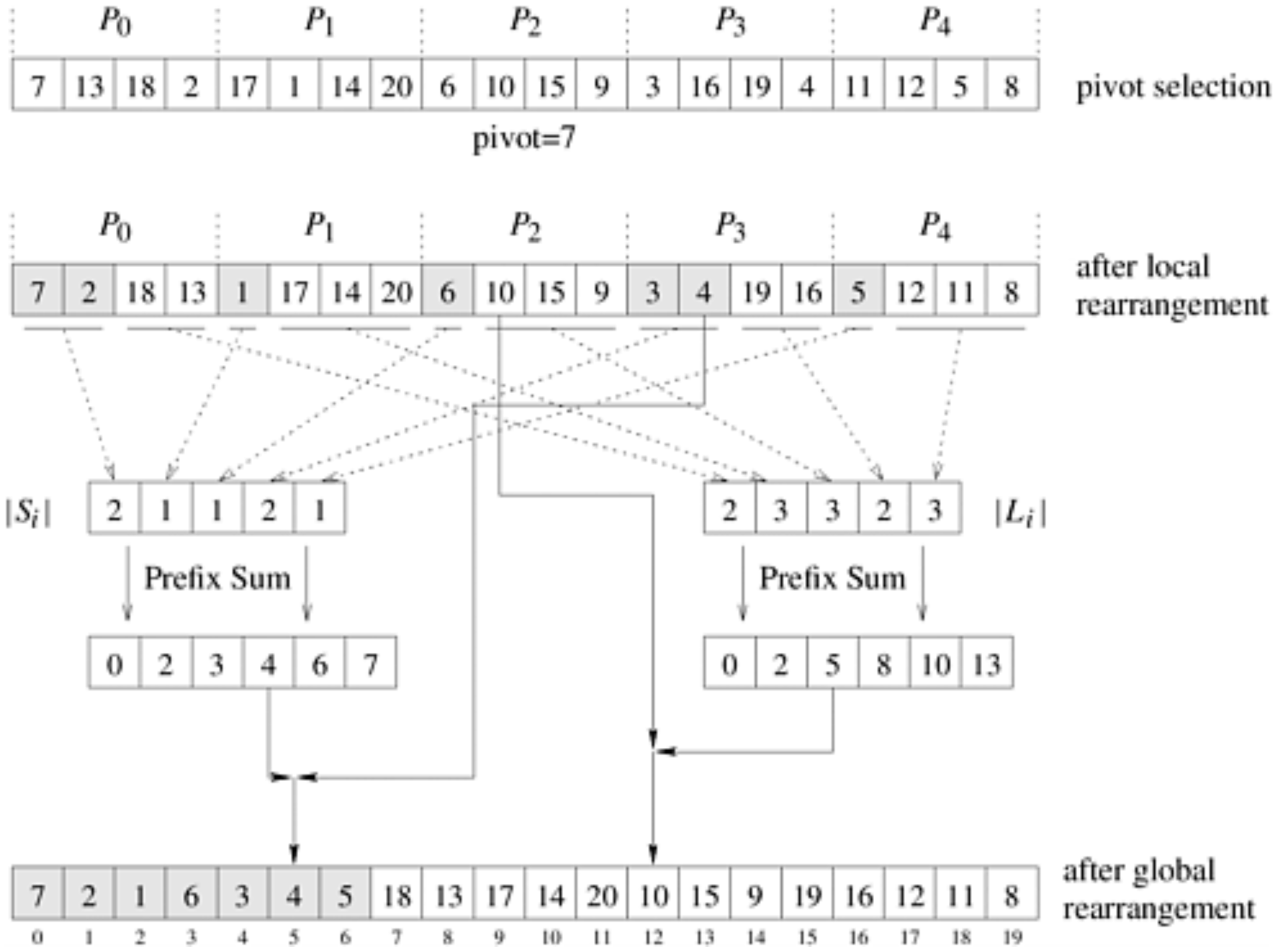
[Musical demo Quicksort ternary]

Some disadvantages

Worst-case running time is $O(n^2)$ when input is already sorted

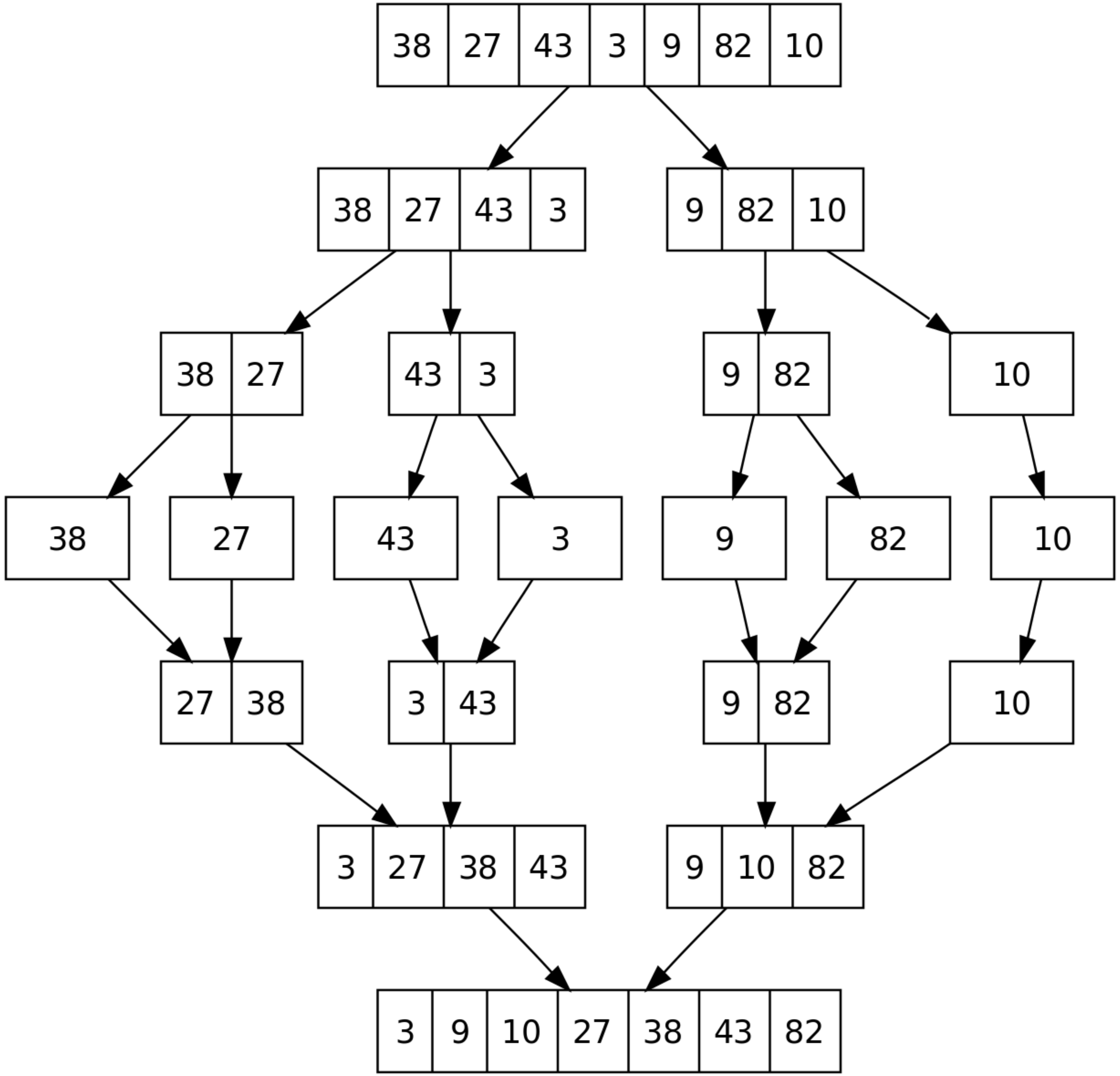
Not stable


First Step
P_0 P_1 P_2 P_3 P_4
7 13 18 2 17 1 14 20 6 10 15 9 3 16 19 4 11 12 5 8
pivot selection
pivot=7
7 2 18 13 1 17 14 20 6 10 15 9 3 4 19 16 5 12 11 8
after local rearrangement
7 2 1 6 3 4 5 18 13 17 14 20 10 15 9 19 16 12 11 8
after global rearrangement
Second Step
7 2 1 6 3 4 5 18 13 17 14 20 10 15 9 19 16 12 11 8
pivot selection
pivot=5
pivot=17
1 2 7 6 3 4 5 14 13 17 18 20 10 15 9 19 16 12 11 8
after local rearrangement
1 2 3 4 5 7 6 14 13 17 10 15 9 16 12 11 8 18 20 19
after global rearrangement
Third Step
1 2 3 4 5 7 6 14 13 17 10 15 9 16 12 11 8 18 20 19
pivot selection
pivot=11
1 2 3 4 5 6 7 10 13 17 14 15 9 8 12 11 16 18 19 20
after local rearrangement
10 9 8 12 11 13 17 14 15 16
after global rearrangement
Fourth Step
10 9 8 12 11 13 17 14 15 16
after local rearrangement
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
Solution


P_0 P_1 P_2 P_3 P_4
7 13 18 2 17 1 14 20 6 10 15 9 3 16 19 4 11 12 5 8
pivot selection
pivot=7
after local rearrangement
7 2 18 13 1 17 14 20 6 10 15 9 3 4 19 16 5 12 11 8
$|S_i|$
2 1 1 2 1
$|L_i|$
2 3 3 2 3
Prefix Sum
0 2 3 4 6 7
0 2 5 8 10 13
after global rearrangement
7 2 1 6 3 4 5 18 13 17 14 20 10 15 9 19 16 12 11 8
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19

Mergesort

1. Subdivide the list into n sub-lists (each with one element).
2. Sub-lists are progressively merged to produce larger ordered sub-lists.


38 27 43 3 9 82 10
38 27 43 3
9 82 10
38 27
43 3
9 82
10
38
27
43
3
9
82
10
27 38
3 43
9 82
10
3 27 38 43
9 10 82
3 9 10 27 38 43 82

Musical demo

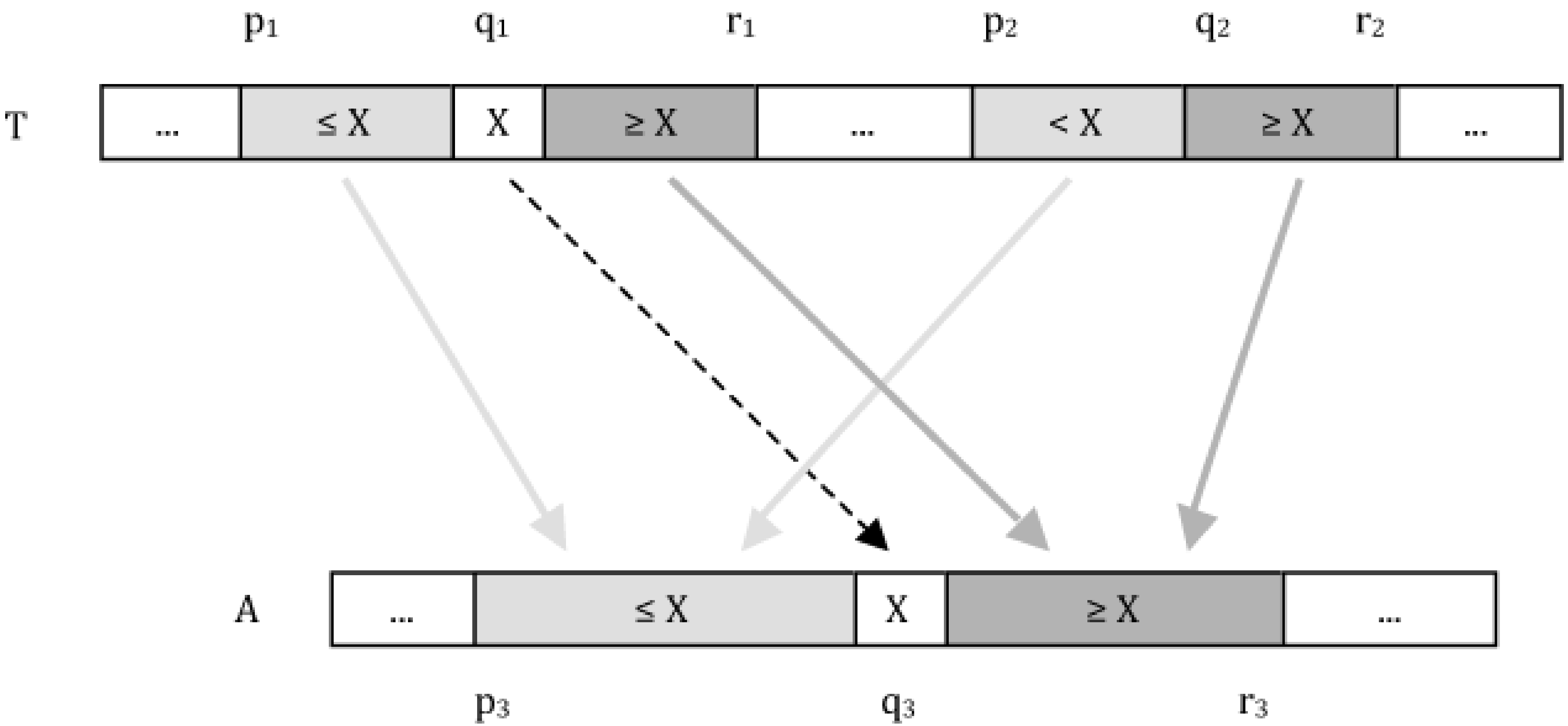
Parallel mergesort

When there are many sub-lists to merge, the parallel implementation is straightforward: assign each sub-list to a thread.

When we get few but large sub-lists, the parallel merge becomes difficult.

Merging large chunks

Subdivide the merge into several smaller merges that can be done concurrently.


p1
q1
r1
p2
q2
r2
T
...
≤ X
X
≥ X
...
< X
≥ X
...
A
...
≤ X
X
≥ X
...
p3
q3
r3

Bucket and sample sort

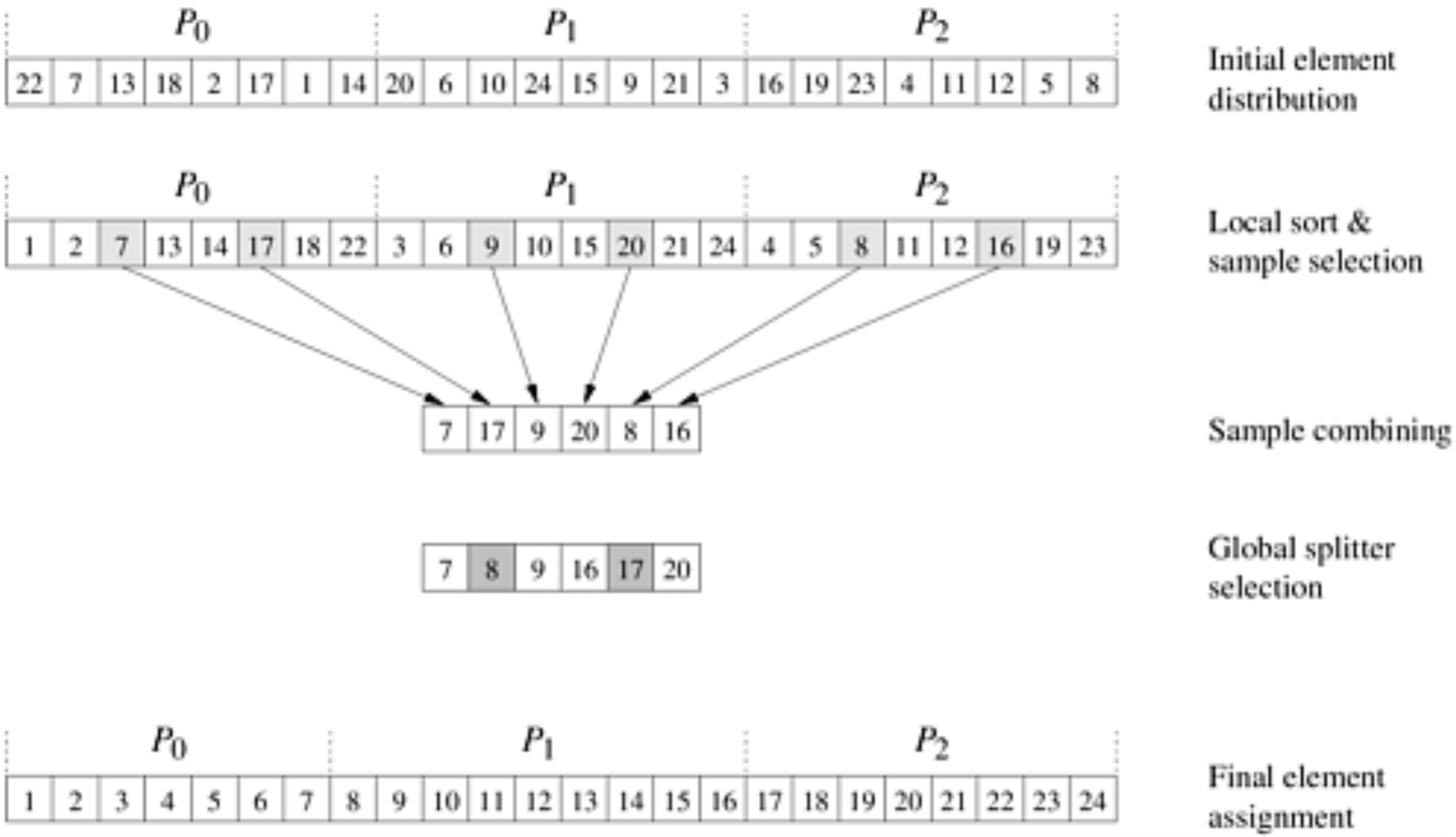
Bucket sort

Sequence of integers in the interval $[a, b]$

1. Split $[a, b]$ into p sub-intervals
2. Move each element to the appropriate bucket (prefix sum)
3. Sort each bucket in parallel!

This process may lead to intervals that are unevenly filled.

Improved version: splitter sort.



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

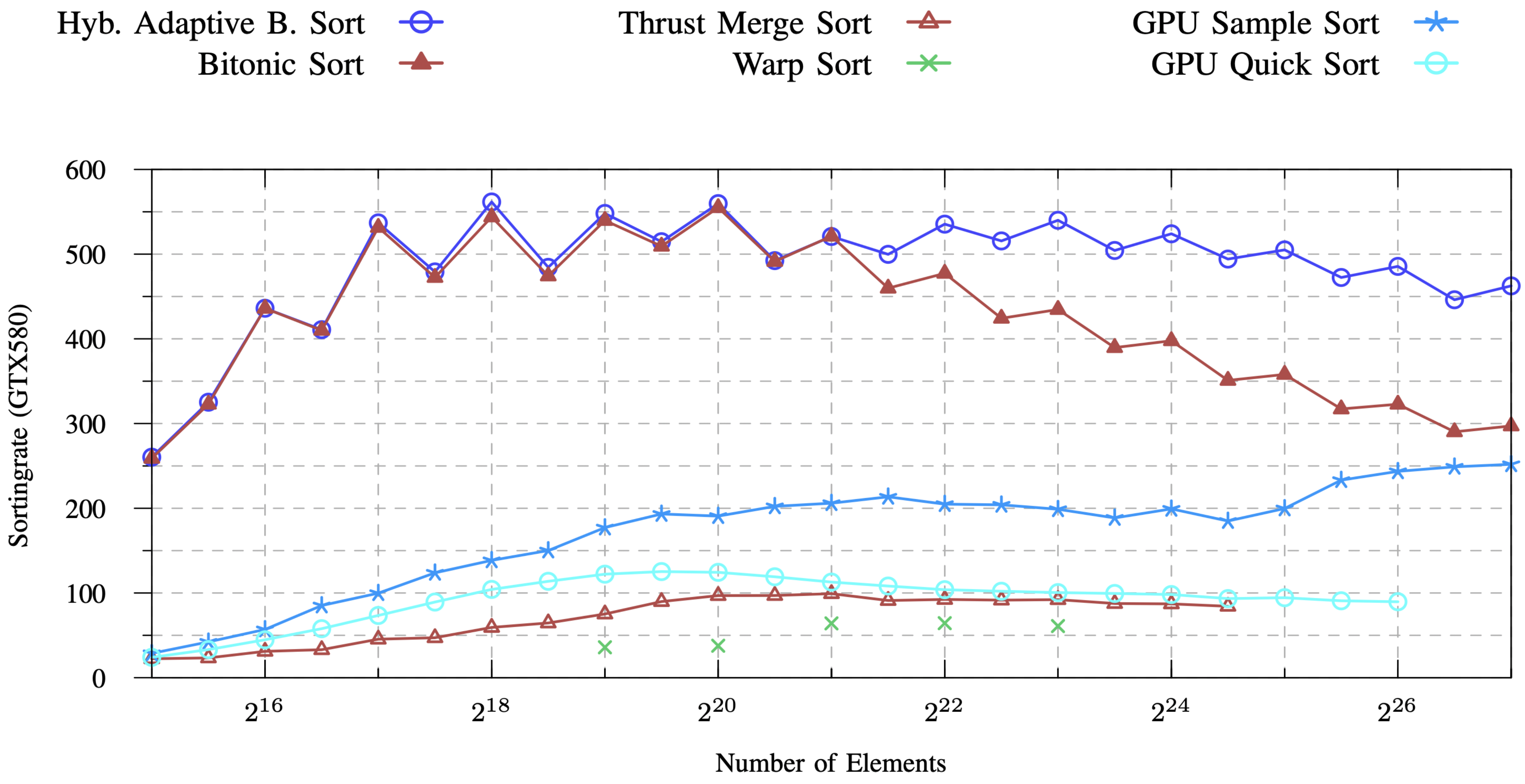
Sorting networks

Building block: compare-and-exchange (COEX)

In sorting networks, the sequence of COEX is **independent** of the data

One of their advantages: very regular data access

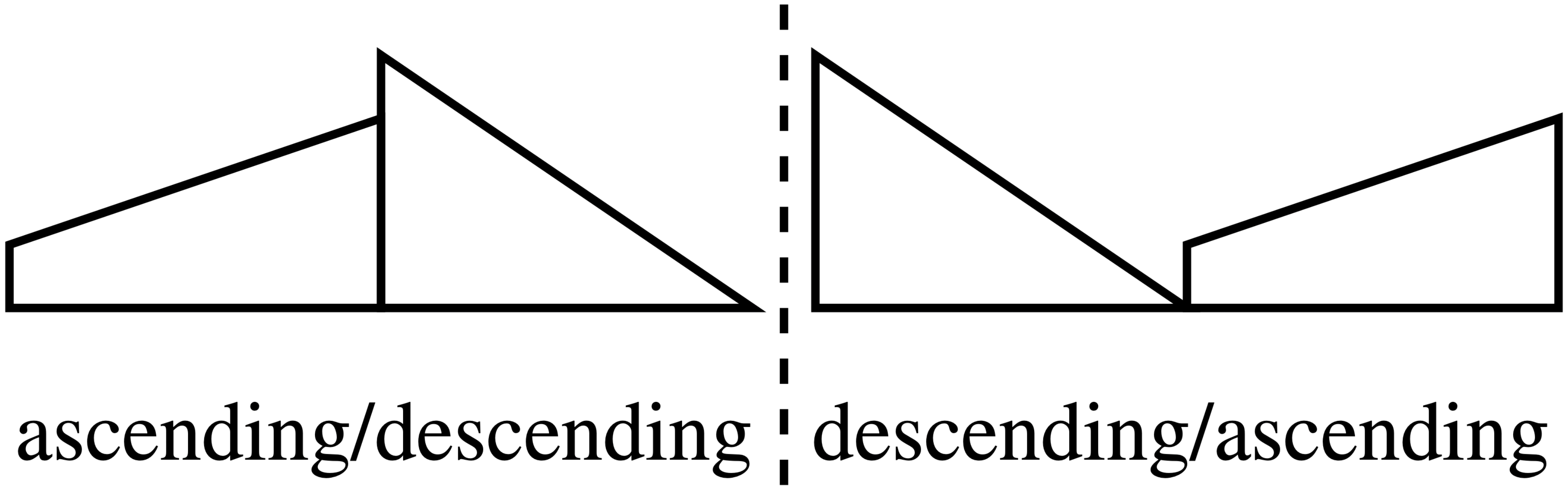
A novel sorting algorithm for many-core architectures based on adaptive bitonic sort, H. Peters, O. Schulz-Hildebrandt, N. Luttenberger



Bitonic sequence

First half ↗, second half ↘, or

First half ↘, second half ↗



There is a fast algorithm to partially "sort" a bitonic sequence

Bitonic *compare*

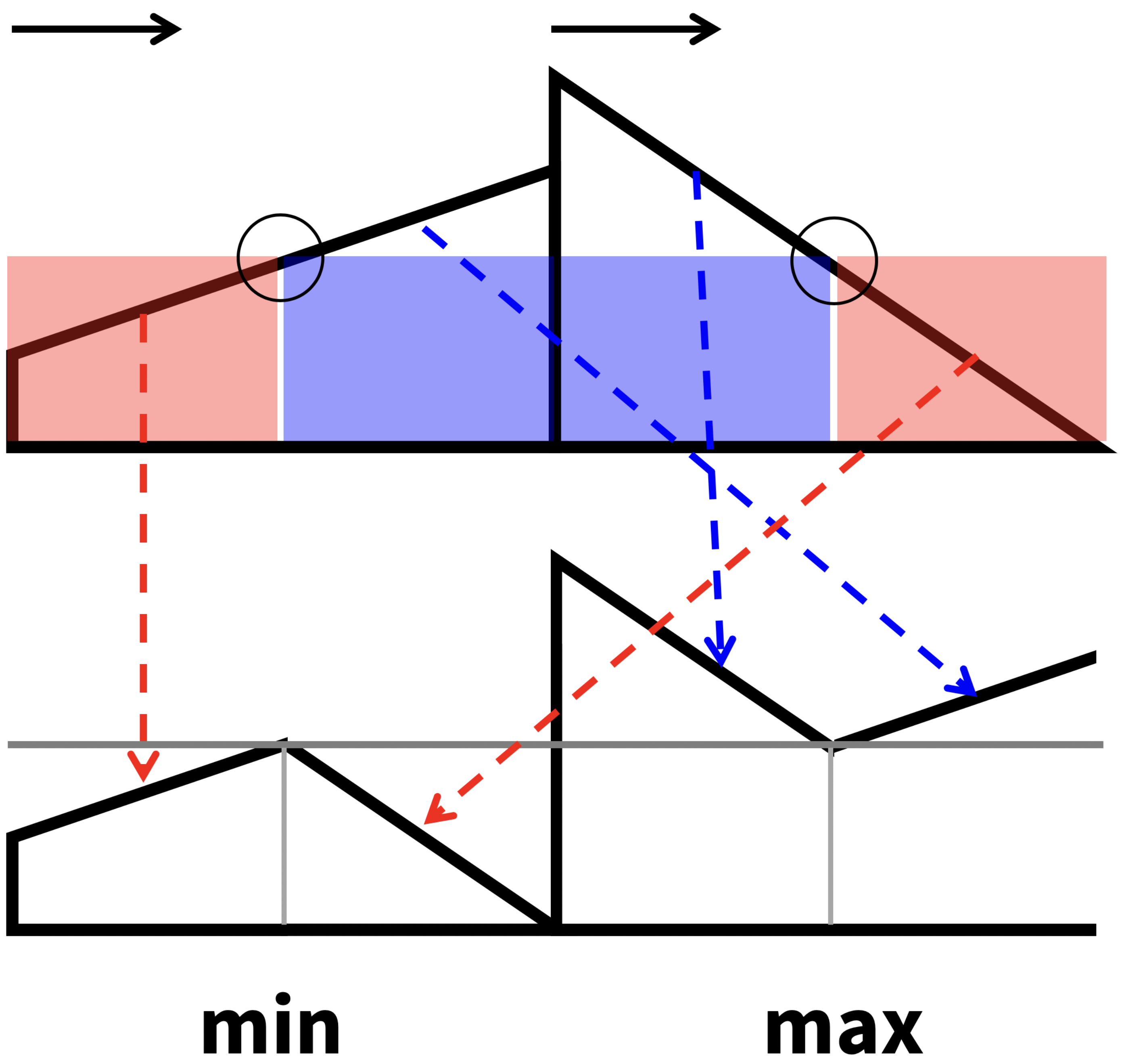
Bitonic compare

First half

$$\min(E_0, E_{n/2}), \min(E_1, E_{n/2+1}), \ldots, \min(E_{n/2-1}, E_{n-1})$$

Second half

$$\max(E_0, E_{n/2}), \max(E_1, E_{n/2+1}), \ldots, \max(E_{n/2-1}, E_{n-1})$$


min
max

Output

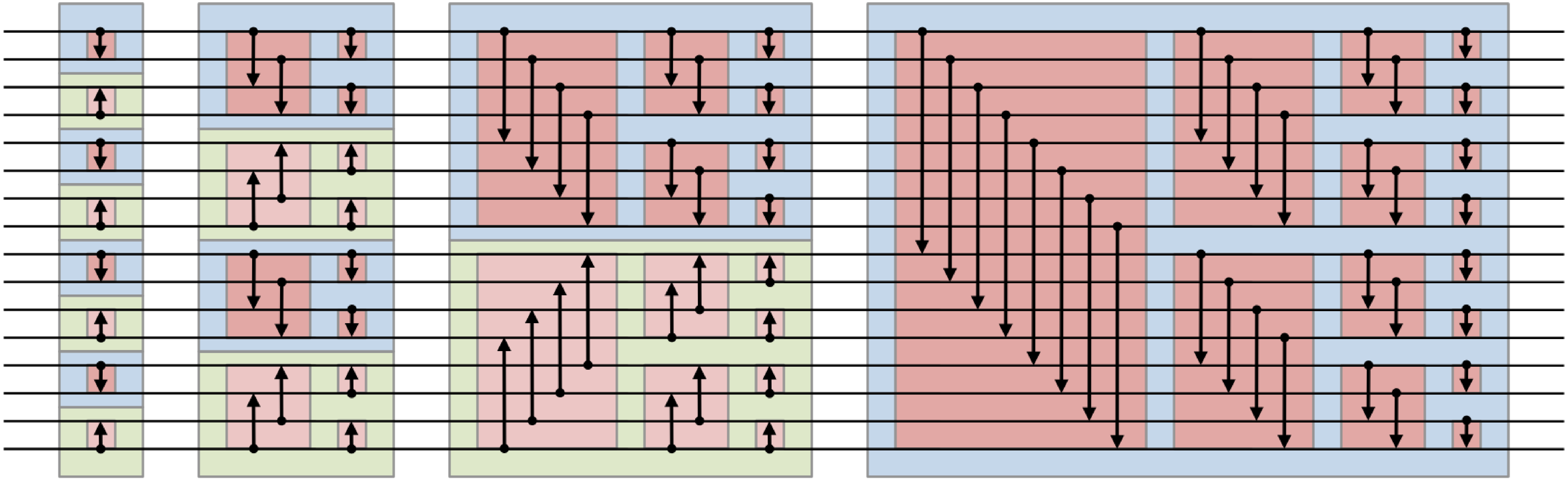
Two bitonic sequences

Left is smaller than right

Build a bitonic sorting network to sort the entire array

Process:

1. Start from small bitonic sequences
2. Use compare and merge to get longer bitonic sequences
3. Repeat until sorted



Complexity

$(\log n)^2$ passes

Musical demo

Python code

Exercise

- `bitonic_sort_lab.cpp` Open this code to start the exercise
- `bitonic_sort.cpp` Solution with OpenMP
- `bitonic_sort_seq.cpp` Reference sequential implementation
- [Code]

`-DNDEBUG` no-debug option

`true` by default

Remove `-DNDEBUG` from `Makefile` to print additional information

Outer `i` loop cannot be parallelized

Step 1: parallelize `j` loop

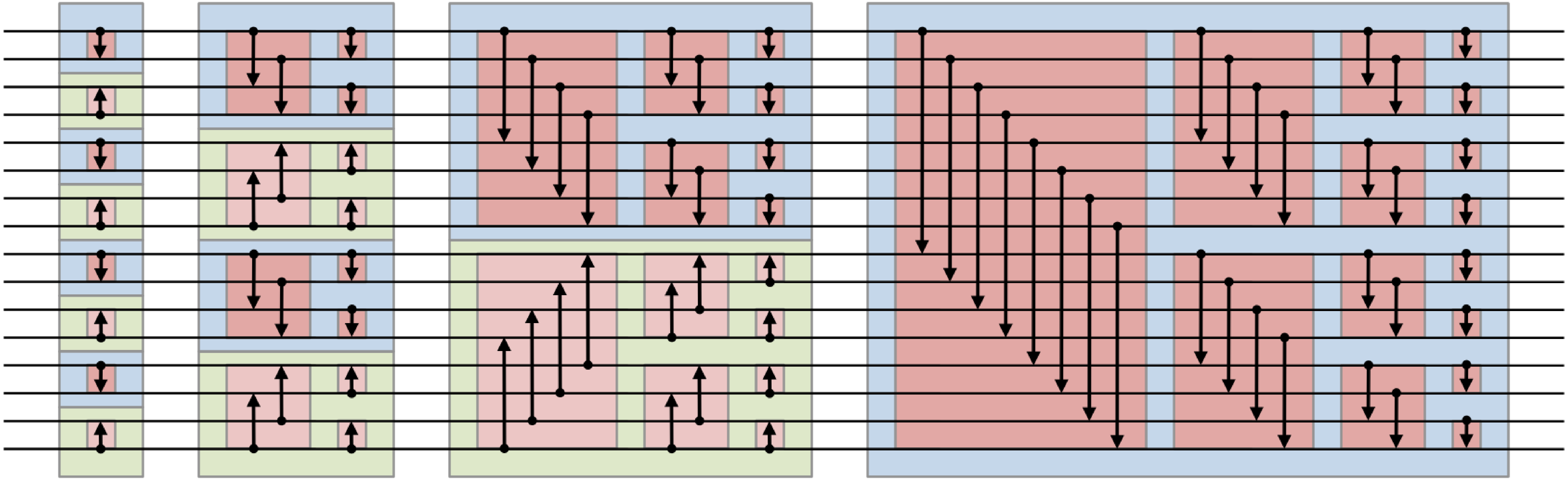
```
for (int j = 0; j < n; j += i)
```

Call `BitonicSortSeq(...)` inside `j` loop

Step 2: split `i` loop into small chunks and large chunks

```
for (int i = 2; i <= chunk; i <<= 1){}
```

```
for (int i = chunk << 1; i <= n; i <<= 1){}
```



Step 3: large-chunk `i` loop

```
for (int i = chunk << 1; i <= n; i <<= 1)
```

Call `BitonicSortPar(j, i, seq, up, chunk)`

`BitonicSortPar()`

`split_length` is very large

Step 4: parallelize `i` loop in `BitonicSortPar()`

```
for (int i = start; i < start + split_length; i++)
```

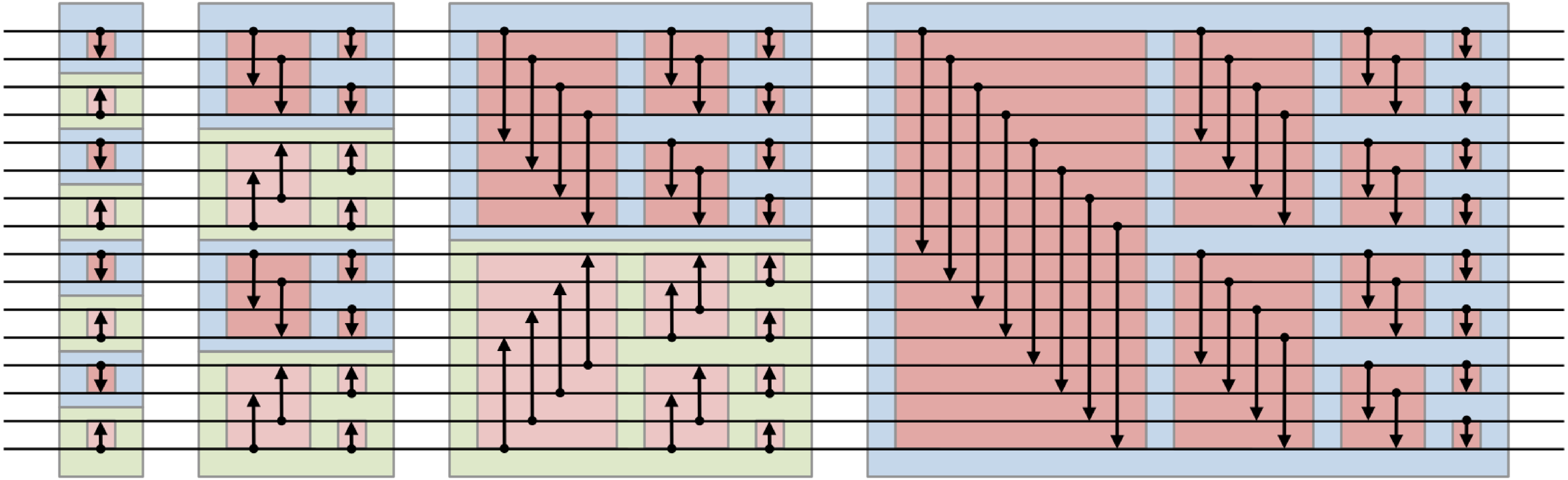
Ultimately fails when `split_length` becomes small again

Step 5: recursively call `BitonicSortPar` only if `split_length > chunk`

Add

```
if (split_length > chunk){}
```

around the two recursive calls to `BitonicSortPar()`



Code is now wrong; one more pass is needed!

Go back to the `i` loop

```
for (int i = chunk << 1; i <= n; i <<= 1){}
```

in `main()`

Step 6: add

```
#pragma omp parallel for
for (int j = 0; j < n; j += chunk)
{
    bool up = ((j / i) % 2 == 0);
    BitonicSortSeq(j, chunk, seq, up);
}
```

at the end of the `i` loop block

```
for (int i = chunk << 1; i <= n; i <<= 1){}
```

The exercise is complete.

Your code should now produce the correct result!

The running time should decrease as you increase the number of threads.

Run using

```
export OMP_NUM_THREADS=4; ./bitonic_sort
```

```
darve@omp:~$ export OMP_NUM_THREADS=1; ./bitonic_sort
Size of array: 8388608
Size of chunks: 8388608
Number of chunks: 1
Number of threads: 1
Elapsed time = 3.24 sec, p T_p = 3.24.
darve@omp:~$ export OMP_NUM_THREADS=4; ./bitonic_sort
Size of array: 8388608
Size of chunks: 2097152
Number of chunks: 4
Number of threads: 4
Elapsed time = 0.83 sec, p T_p = 3.33.
```