CME 213

SPRING 2019

Eric Darve

Shared Memory Sorting

Sorting

- Sorting is a key algorithm in many engineering problems.
- They are both ubiquitous and difficult to implement.
- Sorting is significantly more difficult in parallel than in sequential.
- Algorithms well-suited for sequential sorts are often ill-suited on a parallel machine.

- A fascinating topic!

Quicksort and mergesort

Two great sorting algorithms:

- Occupy a prominent place in world's computational infrastructure.
- Quicksort honored as one of top 10 algorithms of 20th century in science and engineering.

- Quicksort

Quicksort

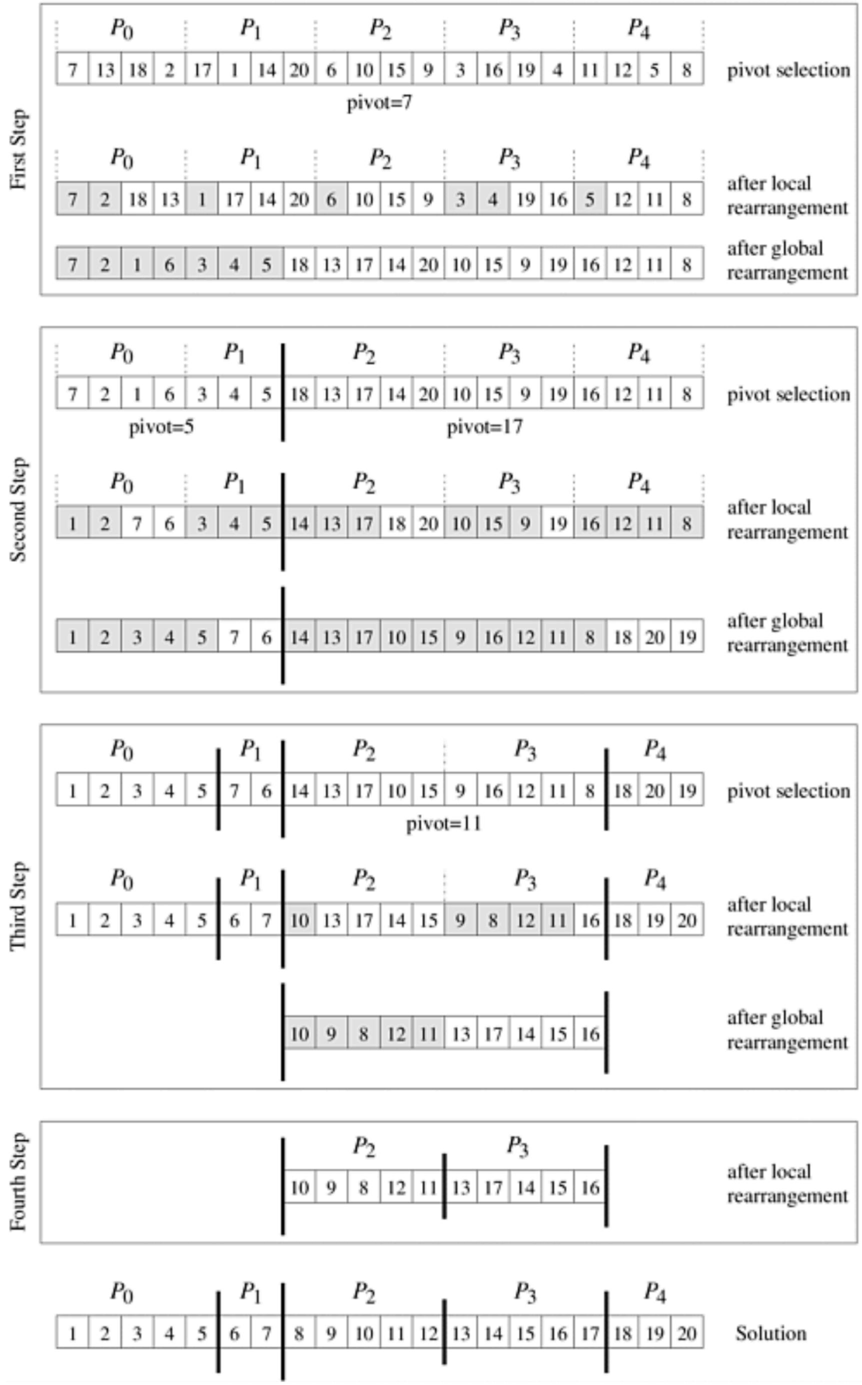
- Most common algorithm.
- Simple, low overhead, optimal average complexity.
- Divide and conquer approach.
- Divide step:
  - Choose a pivot x.
  - Separate sequence into 2 sub-sequences with all elements smaller than x and greater than x.
- Conquer step:
  - Sort the two subsequences.

See Python code `sort.py`

```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:
                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)
```

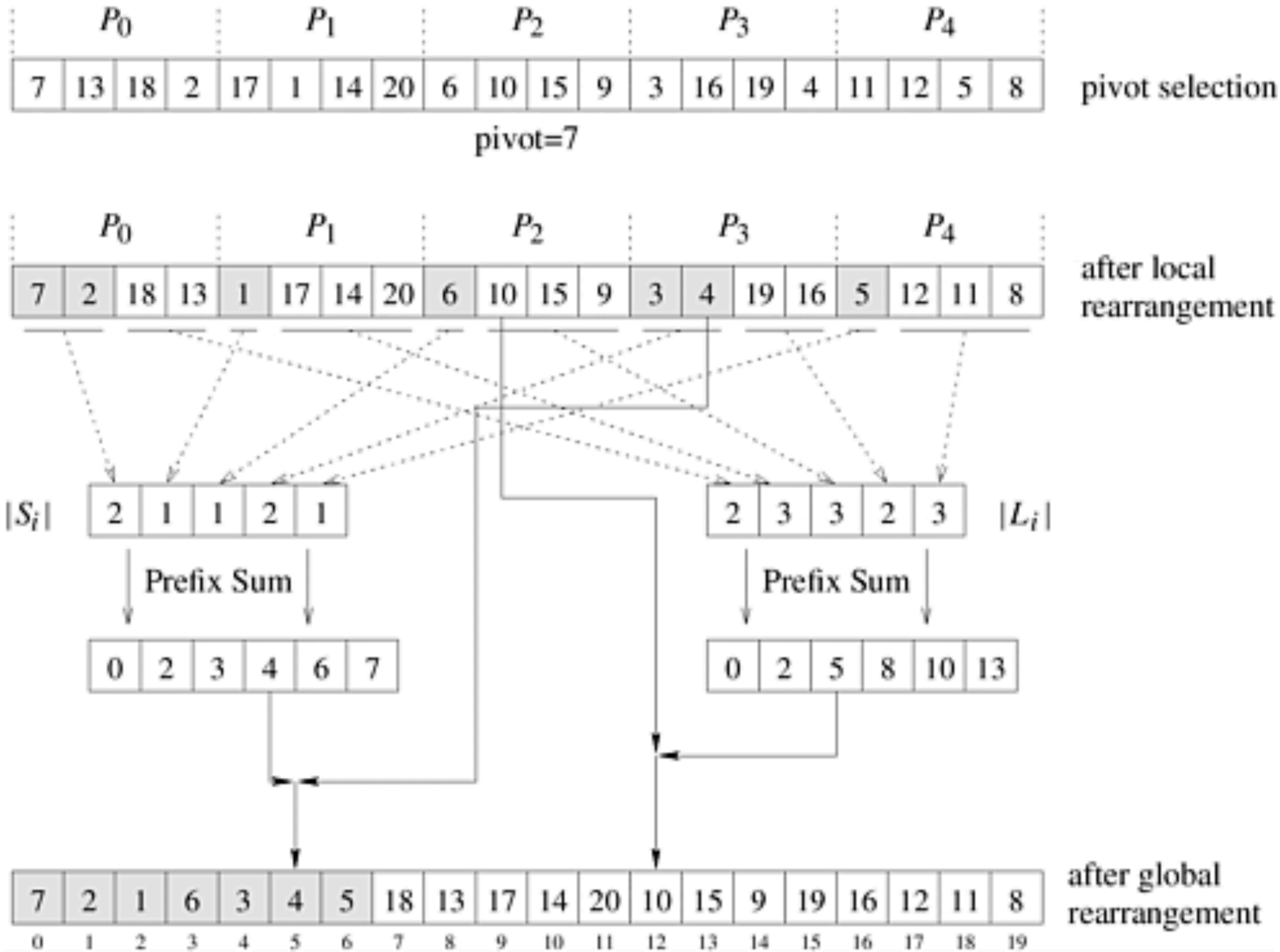
Parallel Quicksort

- Once we have the two sub-lists, we can process each sublist in parallel.
- This works well once the number of sublists is greater than the number of threads.
  - One difficulty is that the length of the sublists can be very different, leading to parallel load-imbalance.
- The creation of the sub-lists in parallel is not easy.
- We split the list into smaller chunks.
- Each chunks is rearranged (pivot rearrangement).
- Then, we compute the global rearrangement.



Prefix sum used to computed the starting addresses for the global rearrangement.

Load imbalance because of sublist sizes and number of processes assigned to each sublist

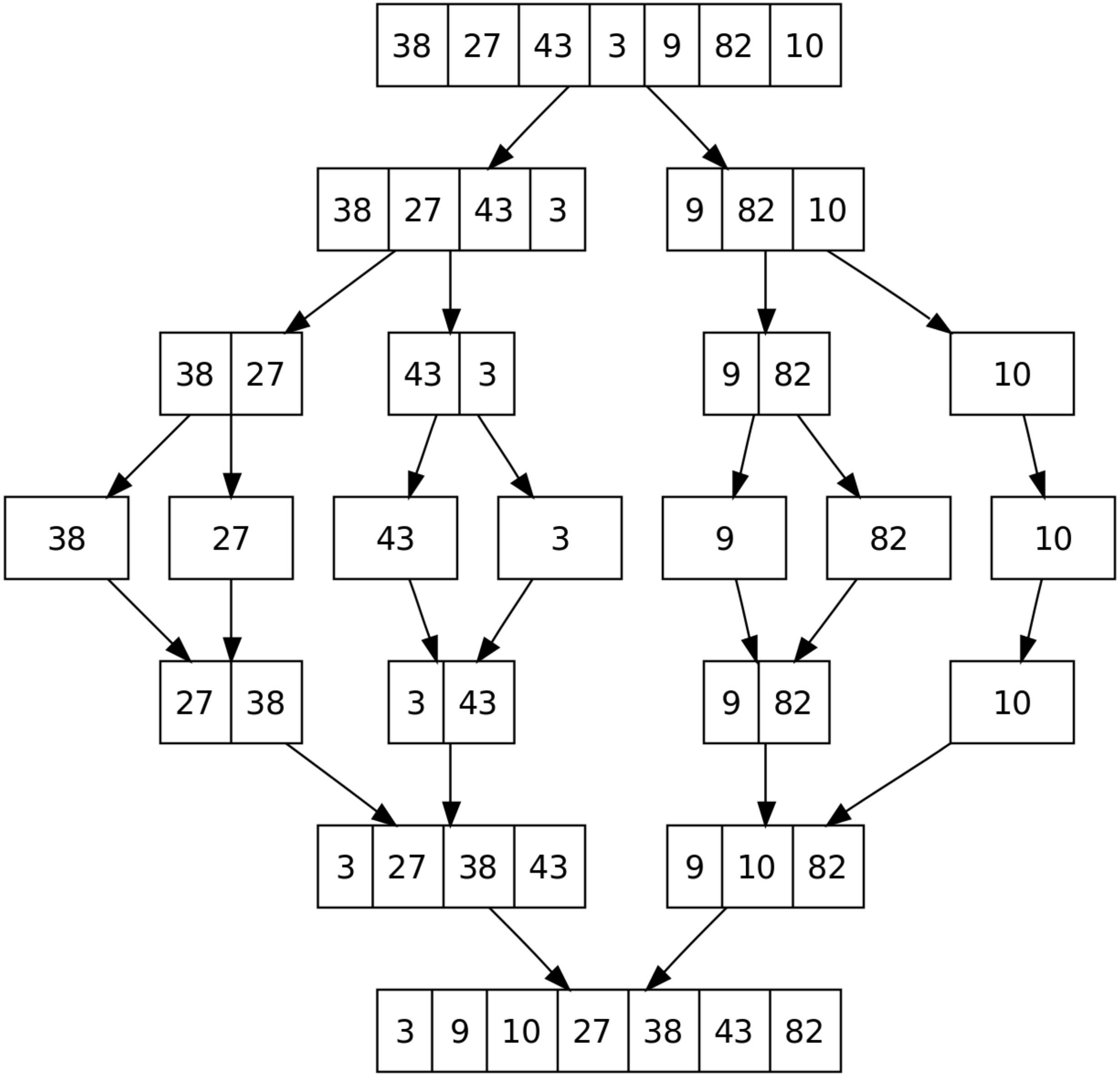


P_0 P_1 P_2 P_3 P_4
7 13 18 2 17 1 14 20 6 10 15 9 3 16 19 4 11 12 5 8
pivot selection
pivot=7
7 2 18 13 1 17 14 20 6 10 15 9 3 4 19 16 5 12 11 8
after local rearrangement
$|S_i|$
2 1 1 2 1
$|L_i|$
2 3 3 2 3
Prefix Sum
Prefix Sum
0 2 3 4 6 7
0 2 5 8 10 13
7 2 1 6 3 4 5 18 13 17 14 20 10 15 9 19 16 12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

Mergesort

- Another popular and classic algorithm.
- We first subdivide the list into n sub-lists (each with one element).
- Then sub-lists are progressively merged to produce larger ordered sub-lists.

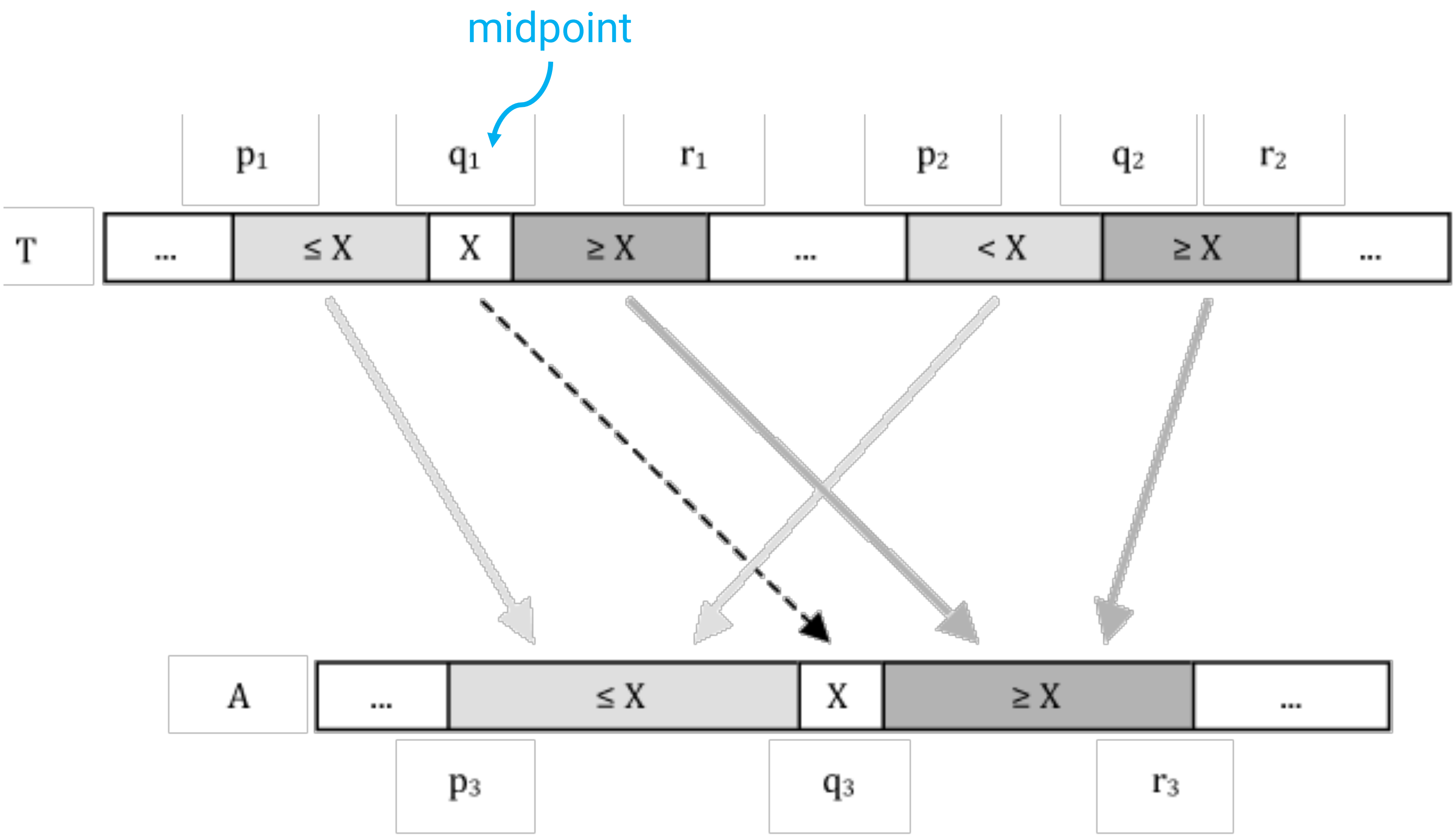
- Merge sort


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

Parallel implementation

- 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 more difficult.
- In that case, we need a way to subdivide the merge into several smaller merges that can be done concurrently.

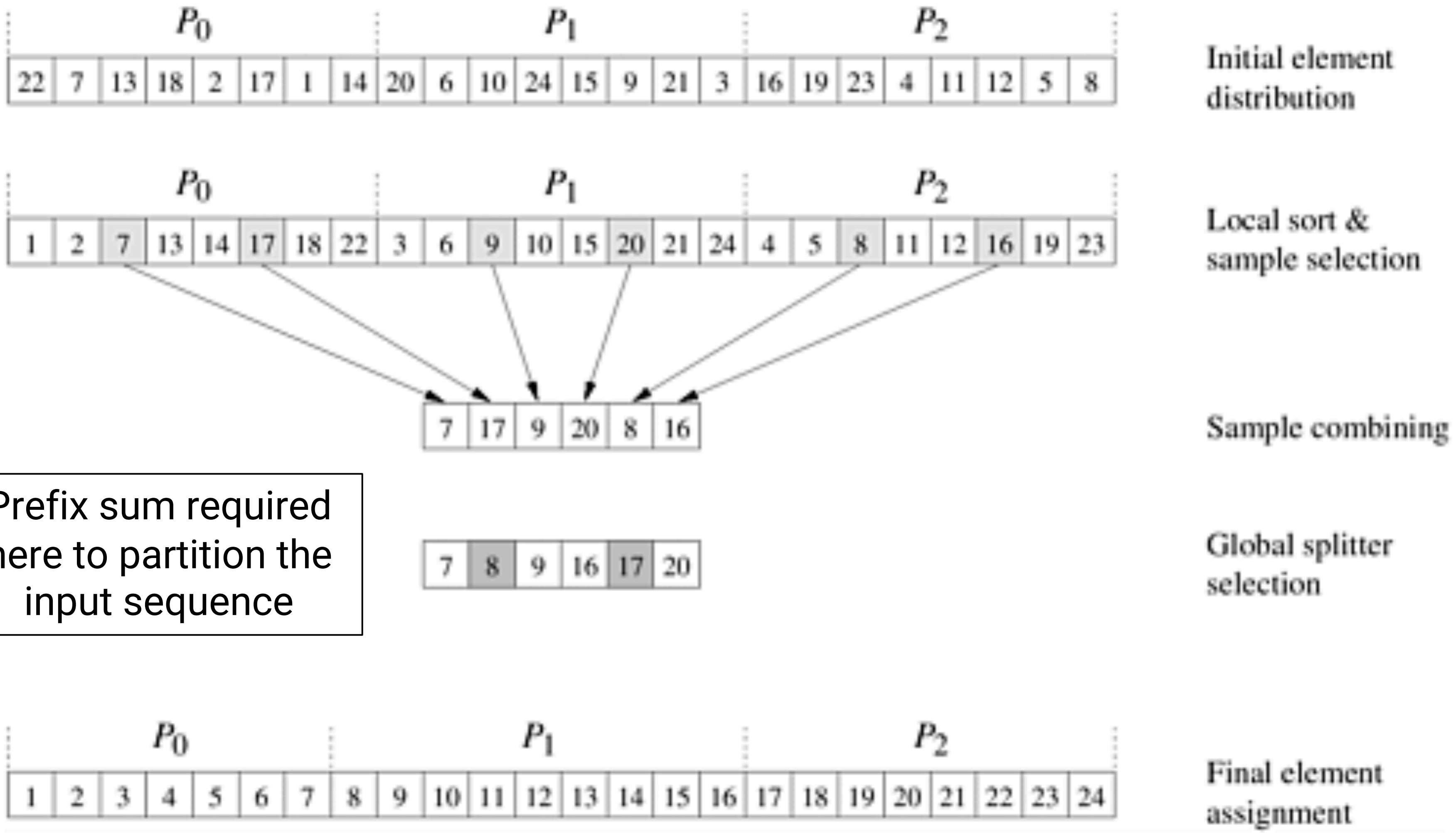
Divide and conquer parallel merge



Big merge replaced by 2 smaller parallel merges

Bucket and sample sort

- Bucket sort is a simpler parallel algorithm.
- Assume we have a sequence of integers in the interval [a,b].
- Split [a,b] into p sub-intervals.
- Move each element to the appropriate bucket (prefix sum required again).
- Sort each bucket in parallel.
- Simple and efficient!
- Radix sort
- Problem: how should we split the interval? This process may lead to intervals that are unevenly filled.
- Improved version: sample (or splitter) sort.



Prefix sum required here to partition the input sequence

Final step is similar to bucket/radix sort

Performance

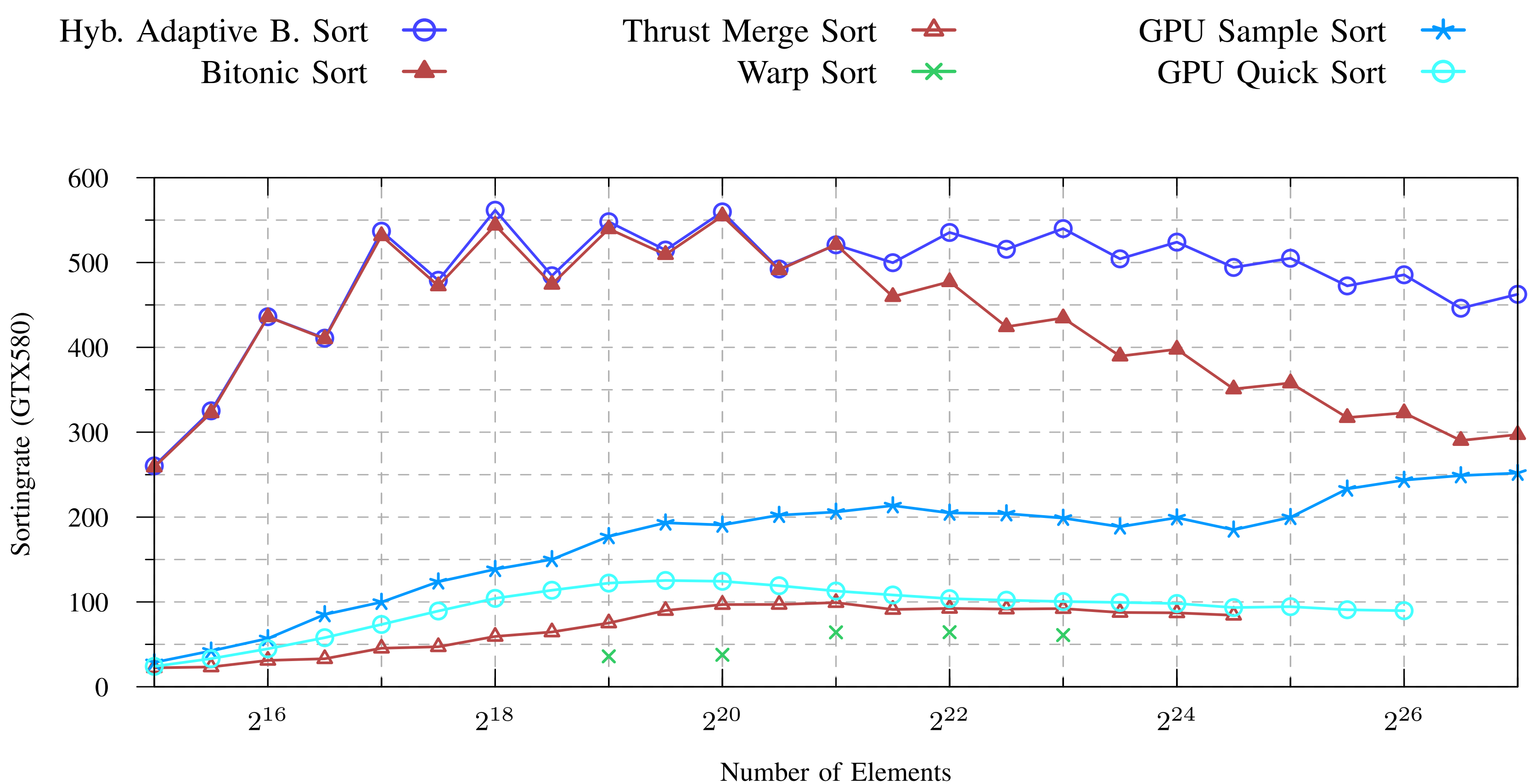
- Sample sort is one of the fastest parallel sorting algorithm.
- One potential bottleneck is sorting the samples (global splitter selection), if this is done sequentially.

Sorting networks

- This is a very special class of sorting algorithms.
- In sorting networks, the sequence of operations (called compare-and-exchange COEX) is independent of the data!
- That is, we perform a deterministic sequence of COEX operations that result in a sorted sequence!
- These algorithms are more complicated but can result in better performance in some cases.
- One of their advantages is that they are very regular compared to the other sorting algorithms.

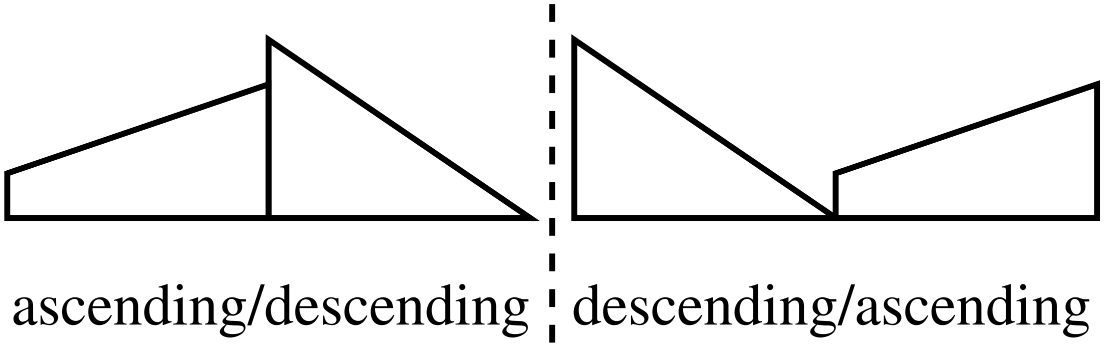
A novel sorting algorithm for many-core architectures based on adaptive bitonic sort

H. Peters, O. Schulz-Hildebrandt, N. Luttenberger



Bitonic sequence

- What is a bitonic sequence?
- A sequence of n numbers such that:
  - The first part is ↗ and the second part is ↘
  - The first part is ↘ and the second part is ↗



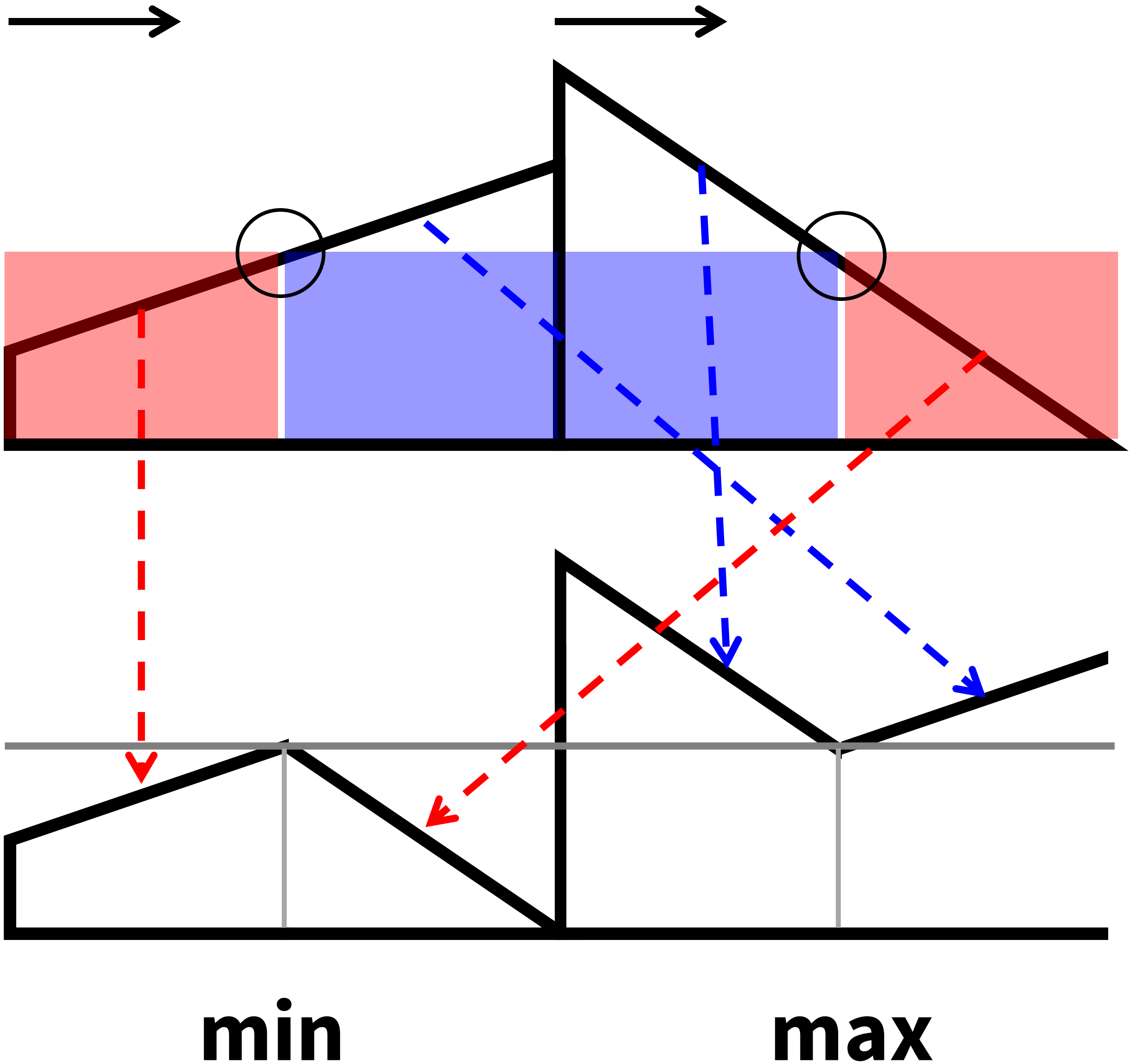
Bitonic merge step

- Consider a bitonic sequence *E* of length *n*.
- Define:

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

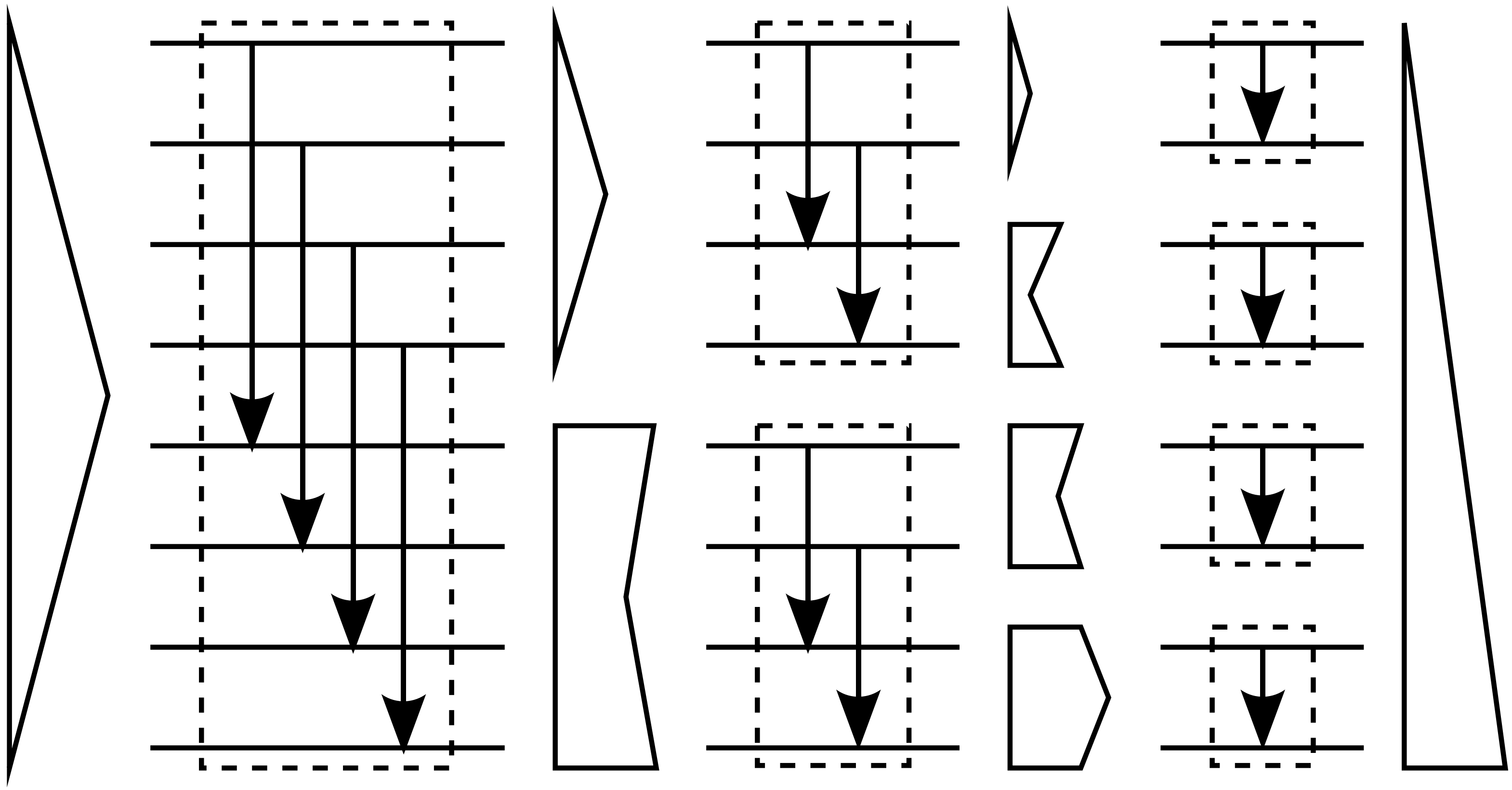
$$U(E) = \Big(\max(E_0, E_{n/2}), \max(E_1, E_{n/2+1}), \ldots, \max(E_{n/2-1}, E_{n-1})\Big)$$

- These two subsequences are bitonic.
- Any element *a* in *L(E)* is smaller than any element *b* in *U(E)*.


min
max

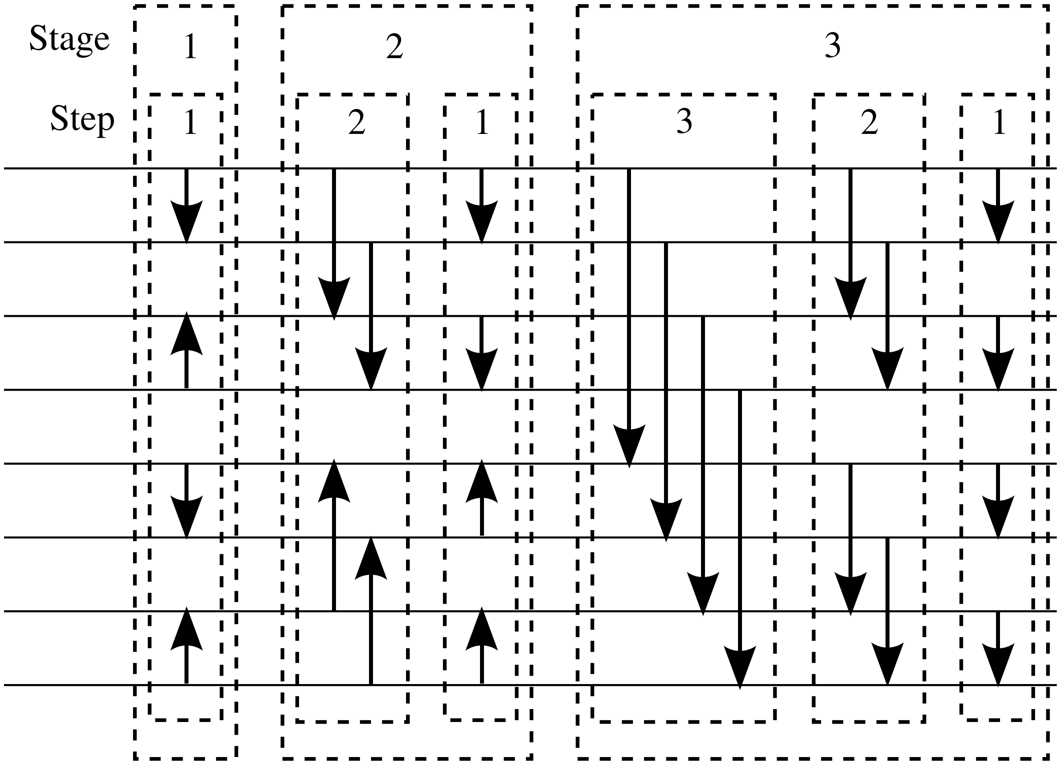
Bitonic merging network

- Start from a bitonic sequence.
- Repeat previous merging process to get a sorted sequence.



Bitonic sorting network

- Start from a sequences of length **2** and sort them.
- Then, sort sequences of lengths **4, 8, 16,** etc, until the whole sequence is sorted.



See Python code `bitonic_sort.py`

[Bitonic]

Computational cost

- This method has cost O(n $\ln^2$ n).
- This cost can be reduced using an adaptive bitonic sort.
- In this variant, the bitonic merge consists of:
    1. Compute how intervals must be rearranged in log(n) steps (for a sequence of size n).
    2. Rearrange the intervals to create two bitonic sub-sequences in log(n) steps using a bitonic tree; log(n) pointers need to be swapped instead of copying data.
- With this algorithm, each merge costs only O(n).
- Total cost is O(n ln n).
- Other optimizations allow making this algorithm much faster than a sample sort on GPUs.

Bitonic sort lab

Different codes are provided:

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

Parallel steps

There are two main parallelization strategies:

1. When we have many bitonic sequences, we can parallelize over the j loop
2. When the bitonic sequences are long, we can parallelize the bitonic merge step (min/max loop).