9. Parallel sorting

- Overview
- Sequential sorting algorithms
- Merging and sorting networks
- Odd-even sorting
- Parallel sample sort

Sequential sorting algorithms

- **Mergesort**
  - Sort recursively and merge two sorted sequences
- **Quicksort**
  - Partition at some pivot, and sort recursively: O(n log n)
- **Radix sort**
  - Sort stably from LSB to MSB
- **Insertion sort, Bubble sort, Macaroni sort, ...**

- **Remember**
  - Comparison-based sorting needs Ω(n logn) steps
  - If we can use additional information on the input, we can do it in O(n)
  - We will look at comparison-based sorting here

Example: Bubblesort

```
Bubblesort(A) :=
  // let A be (a_1, ..., a_n)
  for i in 1..n
    for j in i+1..n
      if (a_{j-1} > a_j)
        swap(a_{j-1}, a_j)
```

Example: Quicksort

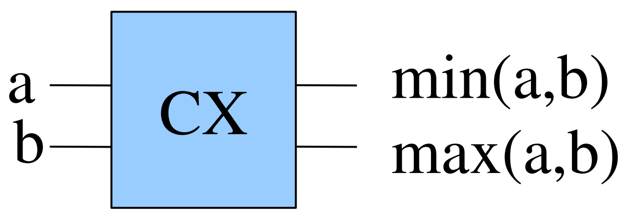
```
Quicksort(A) :=
  if length(A) = 1 then
    return A
  (A1,A2) := Partition(A)
  return Quicksort(A1) + Quicksort(A2)

Partition(A) :=
  choose some pivot p
  return
    ({ a in A | a <= p }, { a in A | a > p })
```

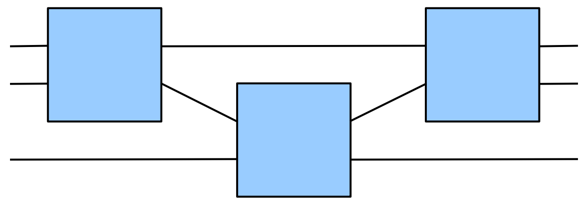
Example: Mergesort

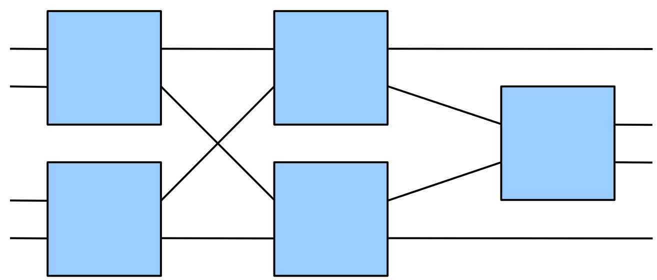
```
Mergesort(A) :=
   // Let A = (a_1, ..., a_n)
   if n = 1
     return A
   m := floor(n/2)
   A1 := MergeSort( (a_1, ..., a_m) )
   A2 := MergeSort( (a_m+1, ..., a_n) )
   Merge(A1,A2)
```

Sorting networks

- Given the importance of sorting, we could try to build it in HW
  - This leads to sorting networks
  - Basic element: compare-and-exchange (CX)
- Examples





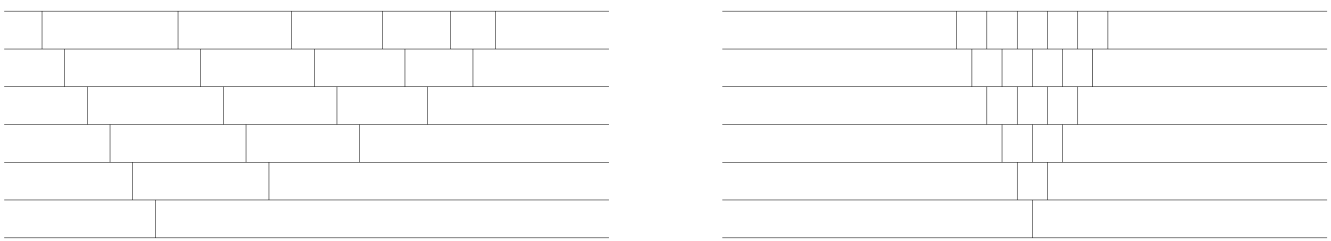


Sorting network

- How do we compare networks?
- A *sorting network* is a DAG, with three kinds of vertices
  - n inputs, of in-degree 0, out-degree 1
  - n outputs of in-degree 1, out-degree 1
  - gates of in- and out-degree 2 (our comparators)
- The *depth* of
  - inputs is 0
  - any gate is 1 + maximum depth of its predecessors
  - outputs is the depth of its predecessors
- The *depth of a network* is the maximum depth over its outputs

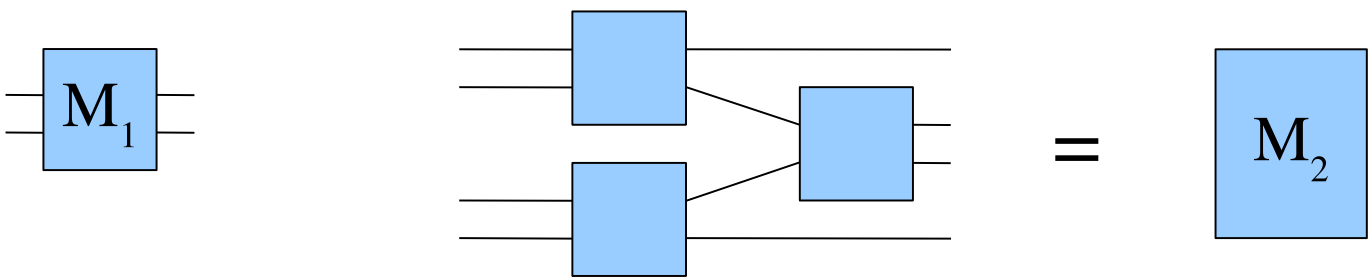
Sorting network

- Implementation of Bubble sort
  - First inner loop can be done with n-1 comparators
  - After that, we sort the remaining n-1 elements
- Cost: $O(n^2)$ comparators, time O(n)
- Can we do better?



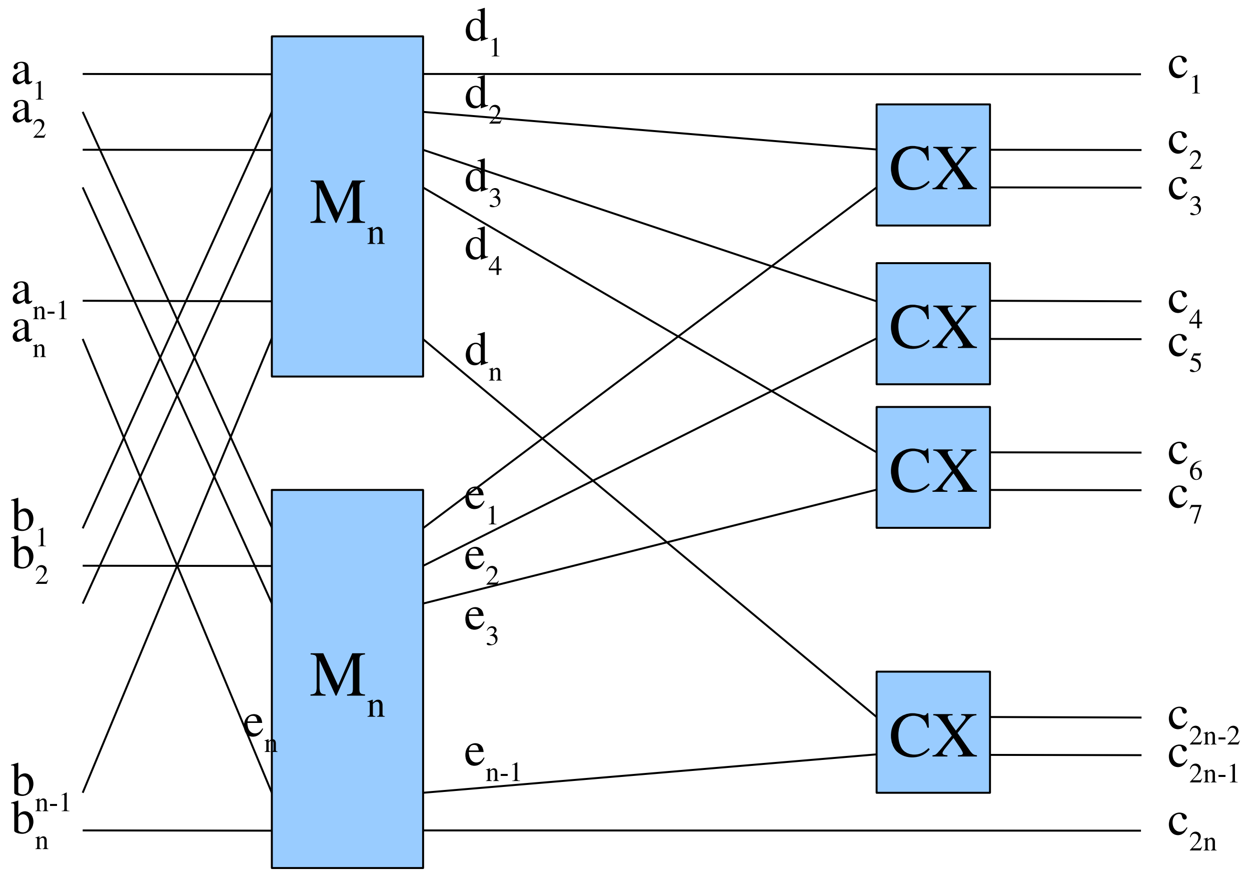
Sorting networks

- **We implement merging**
  - **Assumption: We want to sort 2^n elements**
  - **Start merging sequences of single elements (which are sorted)**
  - **Merge resulting sequences of 2,4,8,... elements up to 2^{n-1}**
- **Merging a small number of elements**



Sorting networks

- **For larger n: we merge recursively**
  - **the odd elements of a and b**
  - **the even elements of a and b**
- **and sort the output**



Sorting networks

- Why does this work?
- Look at element d_i
  - There are at least i-1 odd and i-2 even smaller elements
  - There are at least n-i odd and n-i+1 even larger elements
  - So the final position is 2i-2 or 2i-1
- Look at element e_i
  - There are at least i-1 even and i odd smaller elements
  - There are at least n-i even and n-i-1 odd larger elements
  - So the final position is 2i or 2i+1
- In summary, we know
  - d_1 is the smallest and e_n is the largest element
  - Every pair d_i,e_{i-1}, for 2 <= i <= n compete for positions 2i-2 and 2i-1

Sorting networks

- What does this cost?
- Number of comparators
  - $c(2n) = 2c(n) + n-1$
  - This has solution $c(n)=O(\log n)$
- Time
  - $t(2n)=t(n)+1$
  - This has solution $t(n)=O(\log n)$
- Work
  - $w(n)=c(n)\, t(n) =O(n \log n \log n)$

Sorting networks

- Finally, we can sort: this is called odd-even Mergesort (Batcher 68)
- Combine k = 0, ... , log n -1 stages of mergers
  - Stage k merges 2^{n-k-1} pairs of lists of size 2^k
- Final costs
  - Number of comparators $O(n \log^2 n)$
  - Time $O(\log^2 n)$, Work $O(n \log^4 n)$
- Discussion
  - Excellent speed up, but high number of comparators (more than elements!)
  - Irregular architecture: not feasible to implement for large n
- Observations
  - Networks with time O(log n) and O(n log n) comparators exist
  - This is asymptotically best; but for finite sizes >16, we don't know the optimal networks!

Parallel sort

- Another try at Bubble sort, this time in SW
  - We assume we have n processors, each “owning” an element
  - Then, in each phase
    - » The odd processors execute compare-and-exchange with their right neighbors
    - » Then the even processors do the same
  - In floor(n/2) phases the sequence is sorted
- Cost?
  - Work is $O(n^2)$: not work-optimal
  - Time is $O(n)$: just a log n speedup over Quicksort
- Same as above

Parallel Bubblesort

- Can we do better?
- Idea: Use lesser processors, group the items
  - On p processors, each one works in n/p items
  - Each one locally sorts at start
  - Then each processor does the same as before, but blockwise
  - We have now O(p) phases
- Cost now?
  - Work: $O(p\ (n/p) \log n/p) + O(p\ p\ n/p) = O(n \log n) + O(p\ n)$
  - Time: $O(n/p \log n) + O(n)$
  - Communication: $O(n)$
- This is optimal for $p < \log n$

Sorting by sampling

- Another idea: extend Quicksort
  - Divide input into p partitions
  - Each processors sorts one partition in parallel
  - We assume that each processor has n/p elements at start
- Problems
  - How to partition the data evenly?
  - How to communicate the data afterwards?
- A randomized solution: S*ample sort*

Sample sort: Basic idea

- Draw in parallel a set of k=sqrt(n) random samples $s_1,\ldots,s_k$
  - These will be our pivots
  - Let s_0 be -infinity, s_{k+1} be infinity
- Rearrange all elements in parallel into sqrt(n)+1 buckets
  - Bucket i contains elements in the interval $s_{i-1},\ldots,s_i$
- Sort each bucket in parallel recursively

Theorem (Jaja):

With high probability this terminates in time O(log n) doing O(n log n) operations.

Evident, if distribution into buckets is balanced.

Sample sort: applied to distributed memory

- Each processor i
  - Selects 5 ln n random pivots
  - Sends its pivots to all others
    - » In MPI: an allgatherv operation
  - Sorts the pivots in parallel
  - Chooses positions 5kln n+1, for k=1,...,p-1 as pivots
  - Divides the local n/p elements into p buckets $B_{i1},\ldots,B_{ip}$
  - Sends bucket B_{ij} to processor j
    - » In MPI: an alltoall operation
  - Sorts the local elements (buckets $B_{1i},\ldots,B_{pi}$)

Sample sort: Analysis

Theorem:

With high probability, sample sort terminates in time O(n log n/p) and uses communication time O(p ln n + n/p), for $p^2 <= n/(6 \ln n)$

The key is doing the communication efficiently