Parallel Sorting

John Mellor-Crummey

Department of Computer Science
Rice University

johnmc@rice.edu

Topics for Today

- **Introduction**
- **Sorting networks and Batcher's bitonic sort**
- **Other parallel sorting methods**
  - —sample sort
  - —histogram sort
  - —radix sort
  - —parallel sort with exact splitters

Sorting Algorithm Attributes

- **Internal vs. external**
  - **—internal: data fits in memory**
  - **—external: uses tape or disk**
- **Comparison-based or not**
  - **—comparison sort**
    - **basic operation: compare elements and exchange as necessary**
    - $\Theta(n \log n)$ **comparisons to sort n numbers**
  - **—non-comparison-based sort**
    - ***e.g.* radix sort based on the binary representation of data**
    - $\Theta(n)$ **operations to sort n numbers**
- **Parallel vs. sequential**

Today's focus:
internal parallel comparison-based sorting
distributed memory architectures

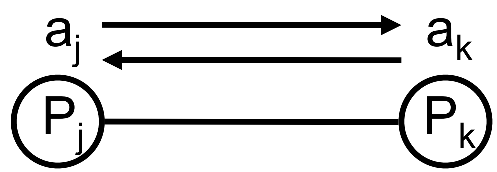
Parallel Sorting Basics

- **Where are the input and output lists stored?**
  - **—both input and output lists are distributed**
- **What is a parallel sorted sequence?**
  - **—sequence partitioned among the processors**
  - **—each processor's sub-sequence is sorted**
  - **—all in** P_j*'s* **sub-sequence < all in** P_k*'s* **sub-sequence if** $j < k$
    - **the best process mapping can depend on network topology**

Element-wise Parallel Compare-Exchange

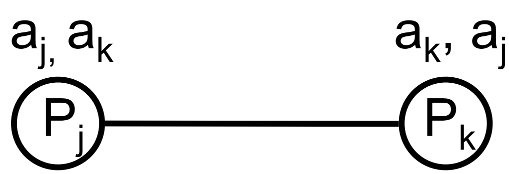
When partitioning is one element per process

1. **Processes P_j and P_k send their elements to each other**

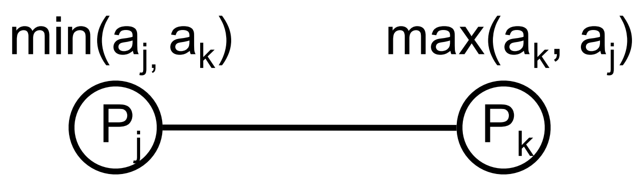


[communication step]

Each process now has both elements

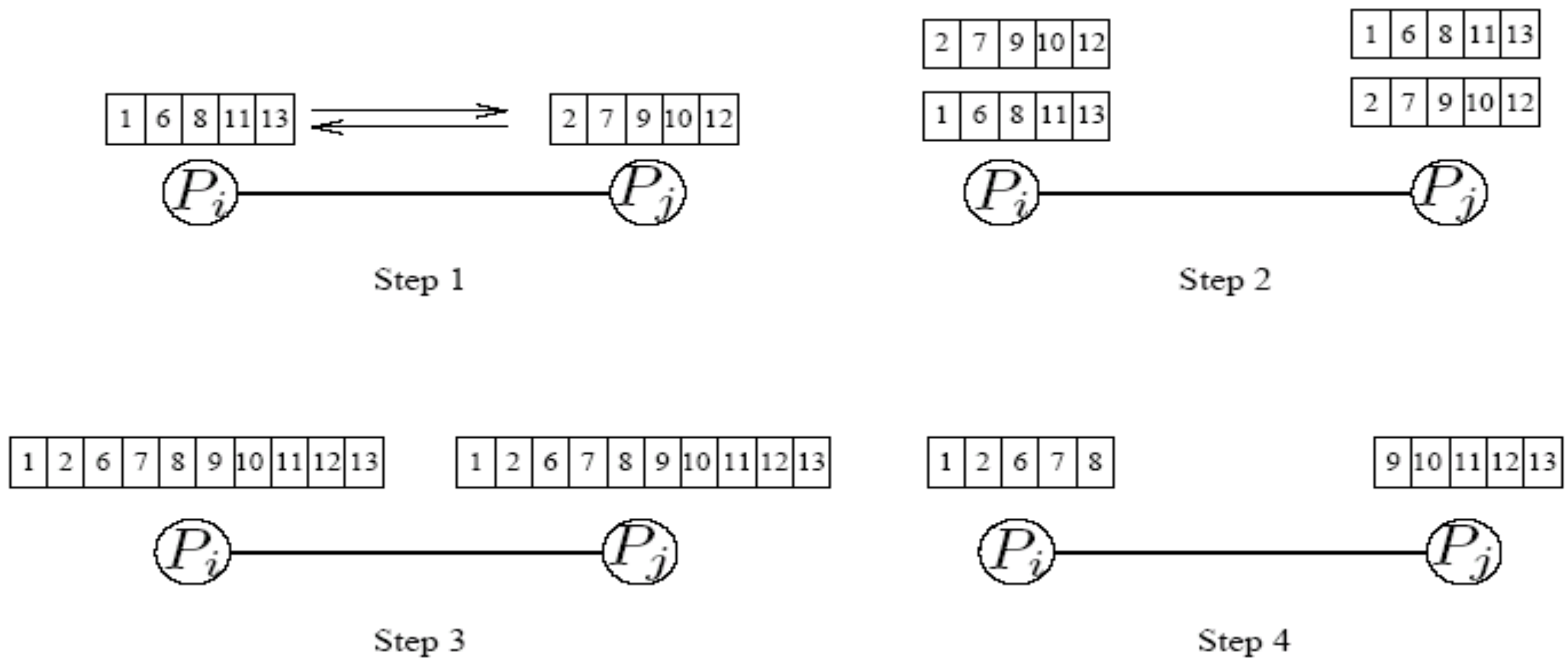


2. **Process P_j keeps $\min(a_j,a_k)$, and P_k keeps $\max(a_j, a_k)$**



[comparison step]

Bulk Parallel Compare-Split

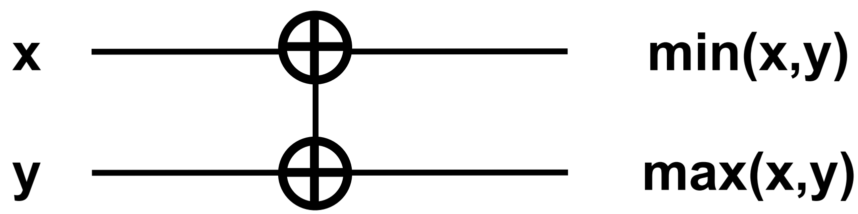


1. **Send block of size *n/p* to partner**
2. **Each partner now has both blocks**
3. **Merge received block with own block**
4. **Retain only the appropriate half of the merged block**

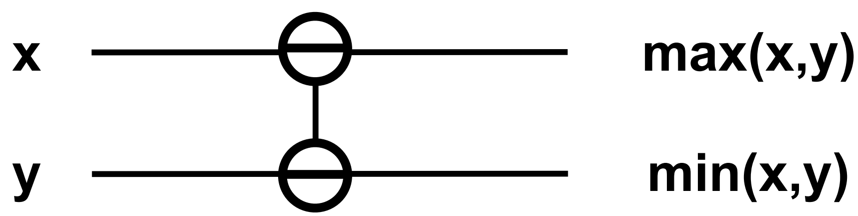
P_i retains smaller values; process P_j retains larger values

Sorting Network

- Network of comparators designed for sorting
- Comparator : two inputs x and y; two outputs x' and y'
  —types
  - increasing (denoted ⊕): x' = min(x,y) and y' = max(x,y)



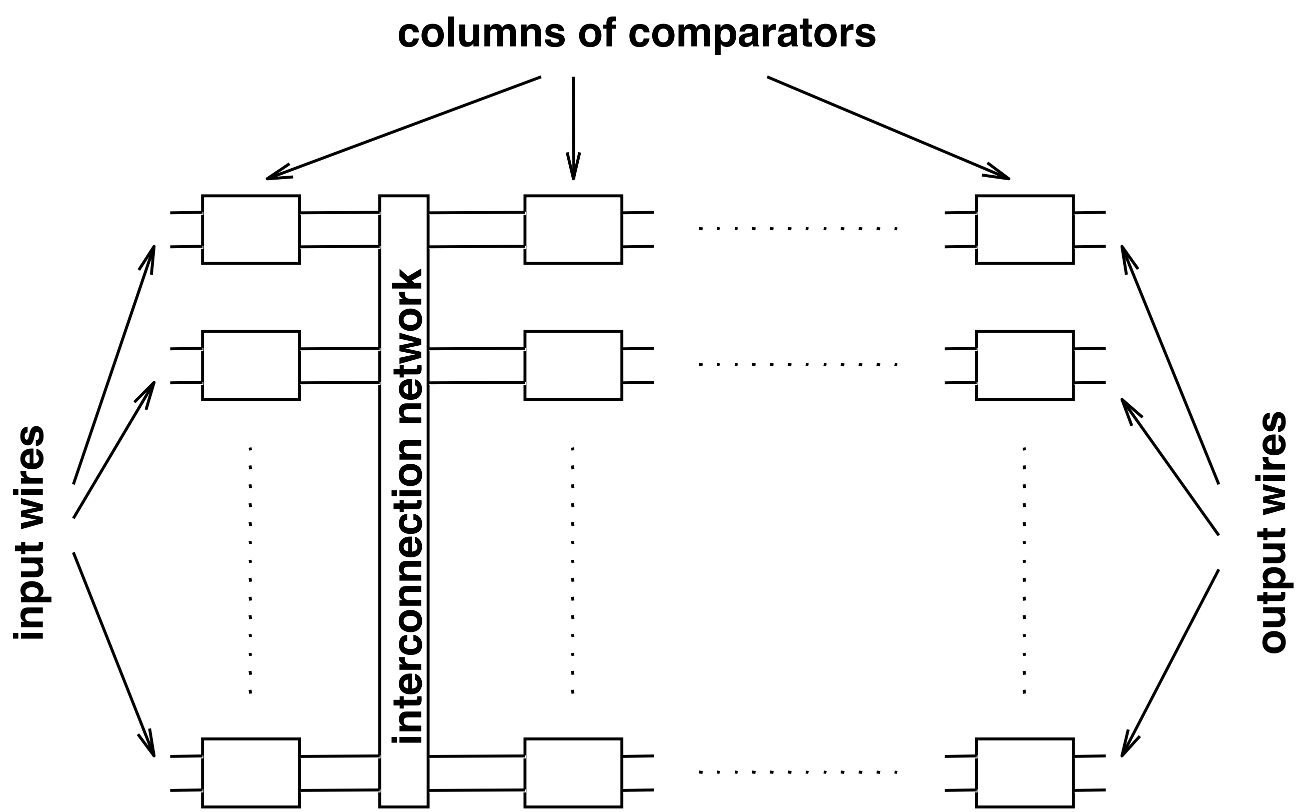
  - decreasing (denoted ⊖) : x' = max(x,y) and y' = min(x,y)



- Sorting network speed is proportional to its depth

Sorting Networks

- Network structure: a series of columns
- Each column consists of a vector of comparators (in parallel)
- Sorting network organization:

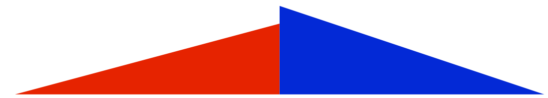


Example: Bitonic Sorting Network

- **Bitonic sequence**
  - **—two parts: increasing and decreasing**
    - **⟨1,2,4,7,6,0⟩: first increases and then decreases (or vice versa)**
  - **—cyclic rotation of a bitonic sequence is also considered bitonic**
    - **⟨8,9,2,1,0,4⟩: cyclic rotation of ⟨0,4,8,9,2,1⟩**
- **Bitonic sorting network**
  - **—sorts n elements in** $\Theta(\log^2 n)$ **time**
  - **—network kernel: rearrange a bitonic sequence into a sorted one**

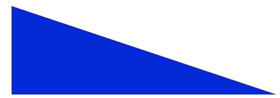
Bitonic Split (Batcher, 1968)

- **Let $s = \langle a_0, a_1, \ldots, a_{n-1} \rangle$ be a bitonic sequence**



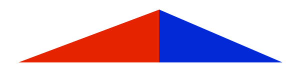
  - $a_0 \leq a_1 \leq \cdots \leq a_{n/2-1}$ **, and**

  - $a_{n/2} \geq a_{n/2+1} \geq \cdots \geq a_{n-1}$

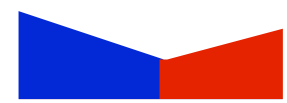


- **Consider the following subsequences of *s***

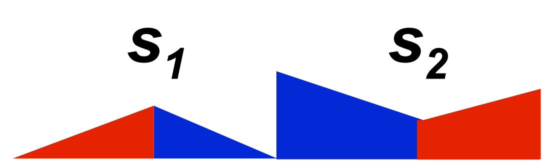
$$s_1 = \langle \min(a_0, a_{n/2}), \min(a_1, a_{n/2+1}), \ldots, \min(a_{n/2-1}, a_{n-1}) \rangle$$



$$s_2 = \langle \max(a_0, a_{n/2}), \max(a_1, a_{n/2+1}), \ldots, \max(a_{n/2-1}, a_{n-1}) \rangle$$

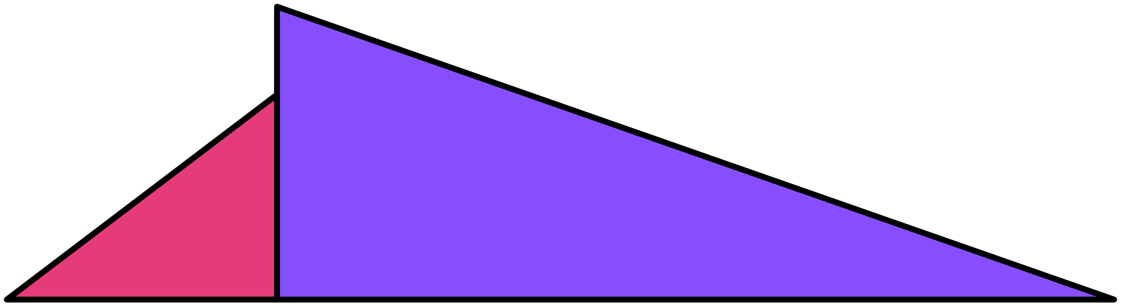


- **Sequence properties**
  - **s_1 and s_2 are both bitonic**
  - $\forall_x \forall_y \; x \in s_1, y \in s_2, x < y$



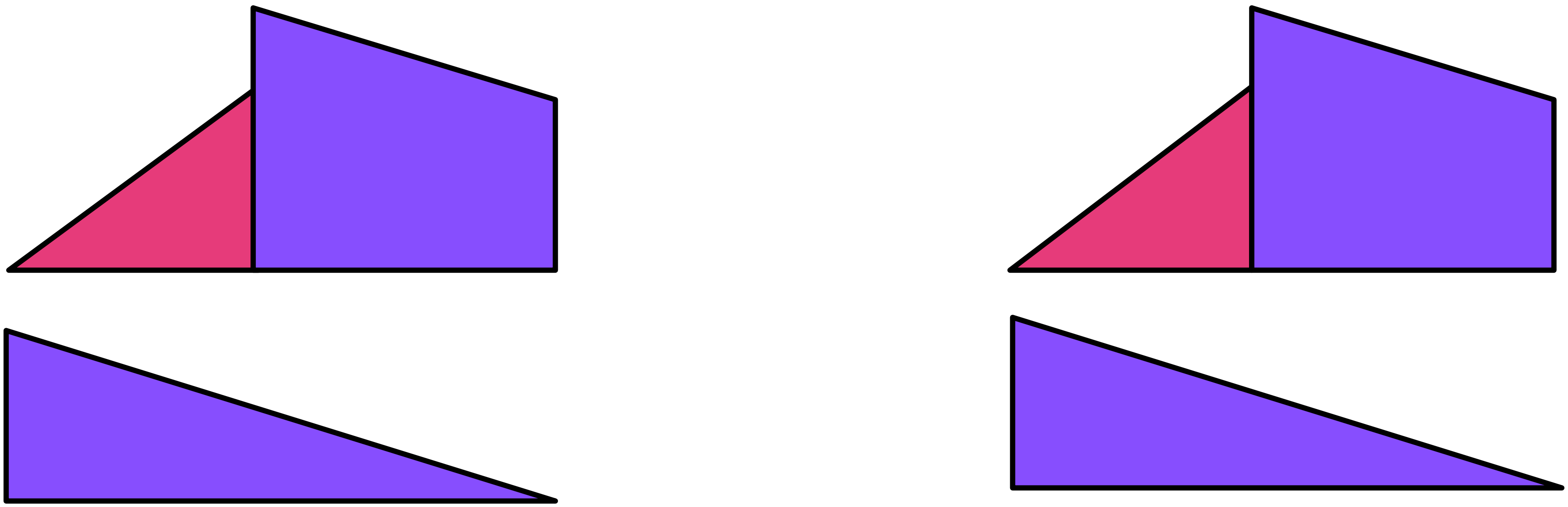
- **Apply recursively on s_1 and s_2 to produce a sorted sequence**
- **Works for any bitonic sequence, even if the increasing and decreasing parts are different lengths**

Splitting Bitonic Sequences - I



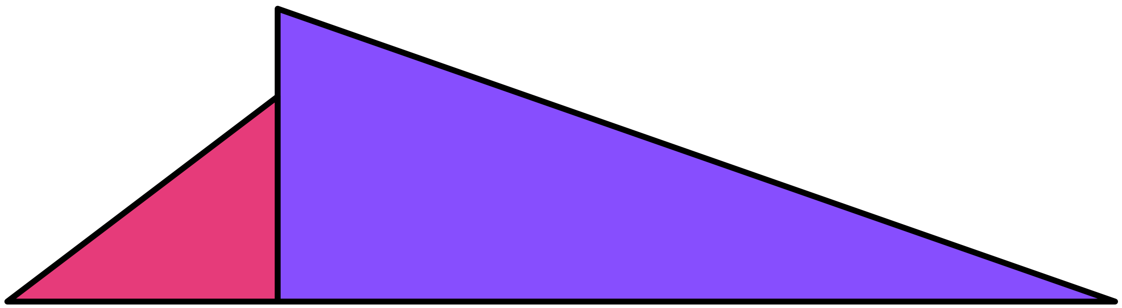
Sequence properties

- s_1 and s_2 are both bitonic
- $\forall_x \forall_y \; x \in s_1, y \in s_2, x < y$



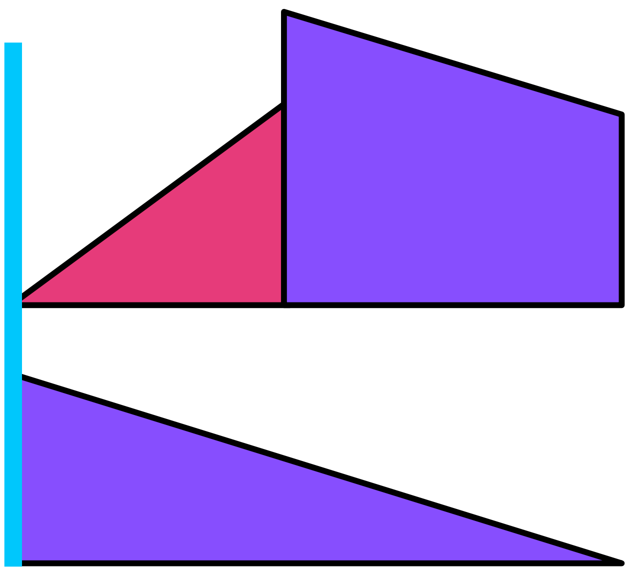
min max

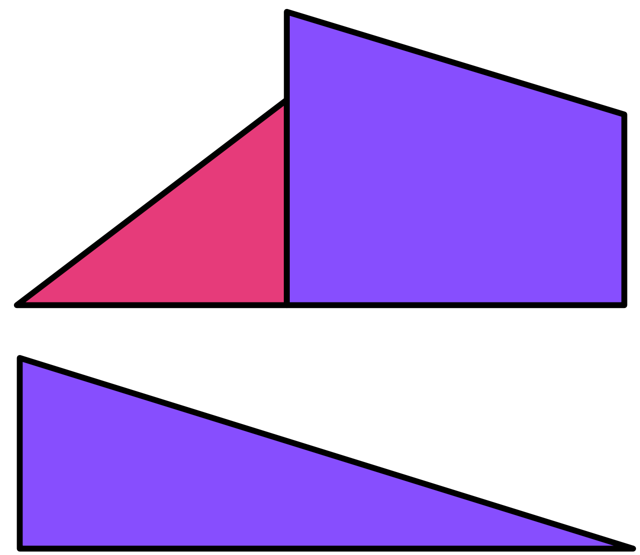
Splitting Bitonic Sequences - I



Sequence properties

- s_1 and s_2 are both bitonic
- $\forall_x \forall_y \; x \in s_1, y \in s_2, x < y$





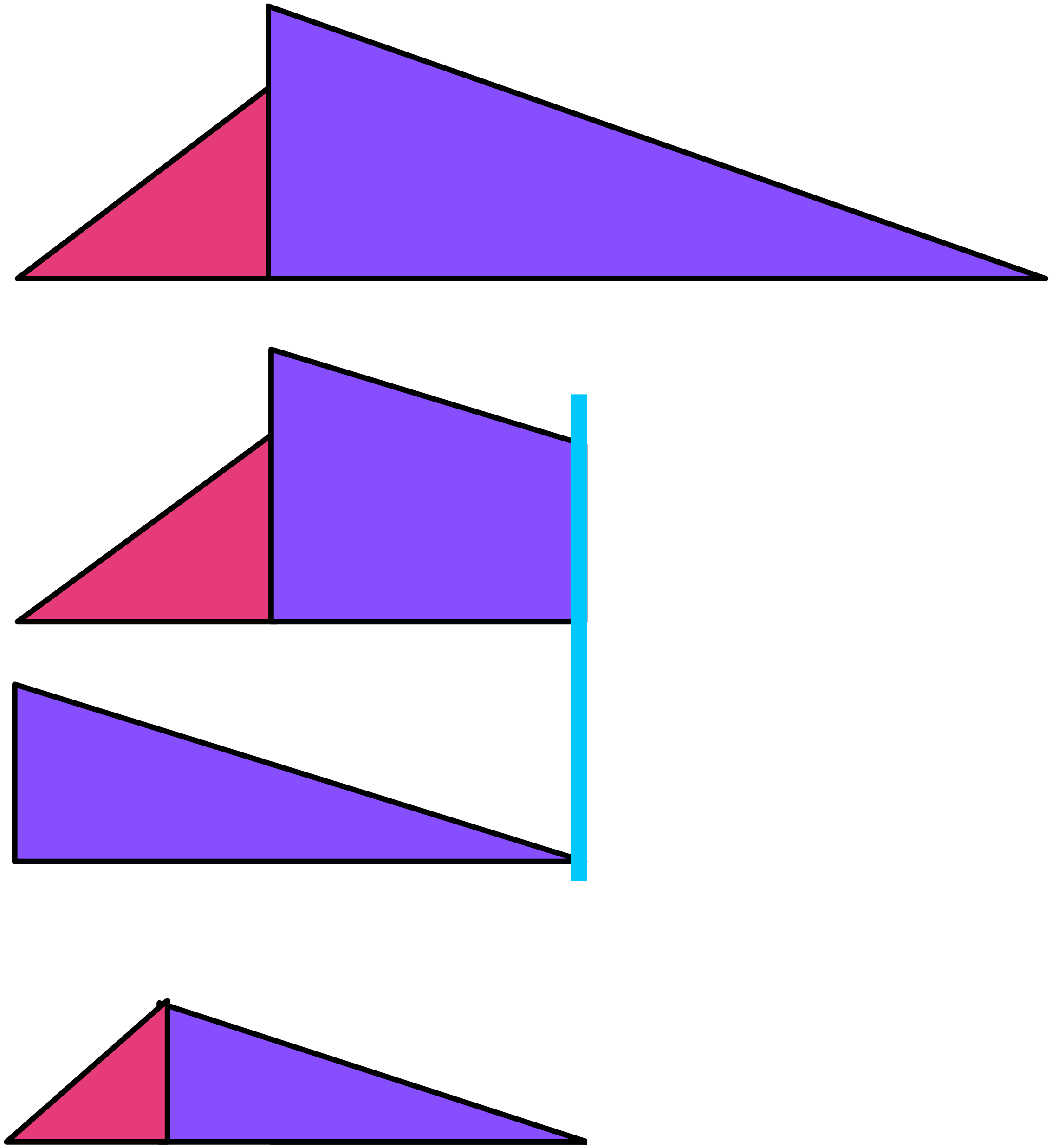
min

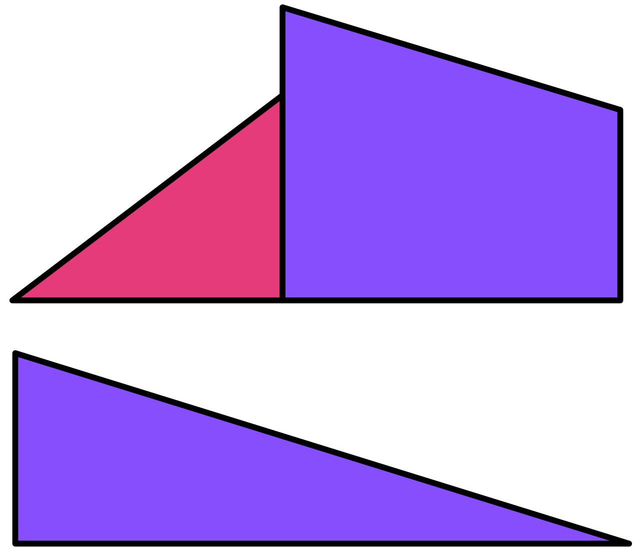
max

Splitting Bitonic Sequences - I

Sequence properties

- s_1 and s_2 are both bitonic
- $\forall_x \forall_y \; x \in s_1, y \in s_2, x < y$

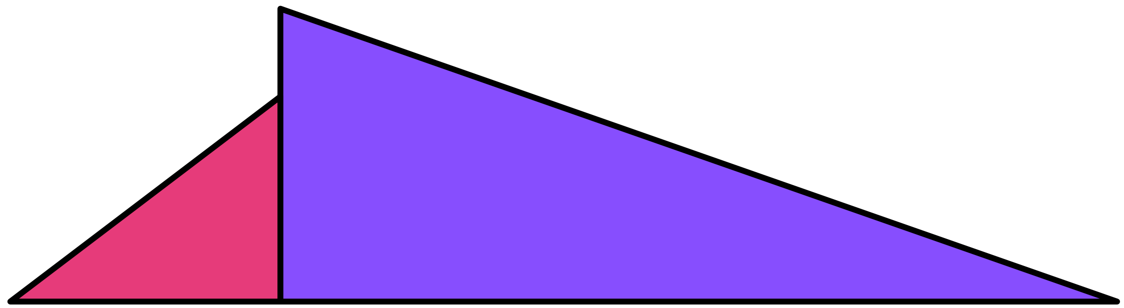




min

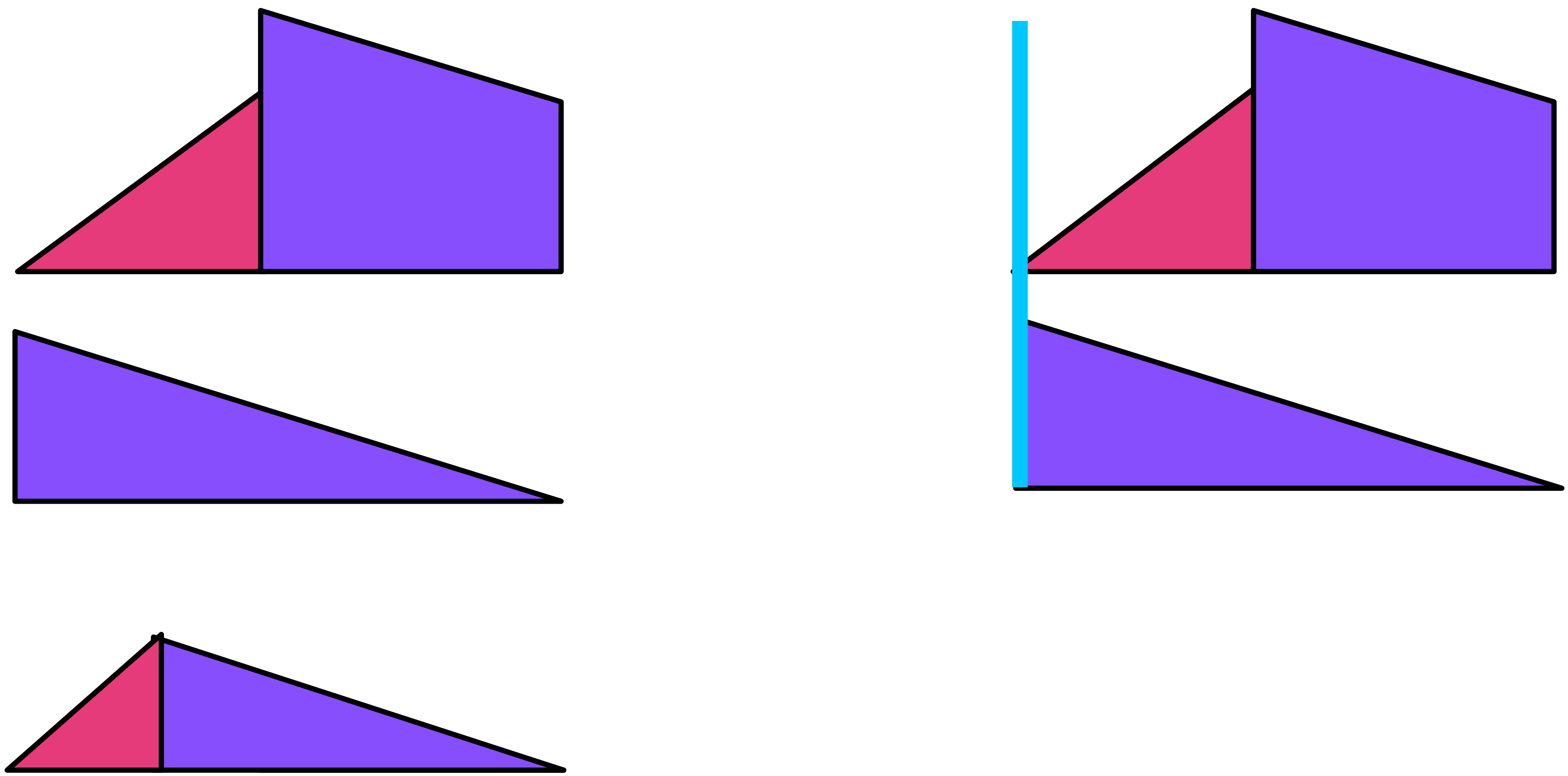
max

Splitting Bitonic Sequences - I



Sequence properties

- s_1 and s_2 are both bitonic
- $\forall_x \forall_y \; x \in s_1, y \in s_2 \,,\; x < y$

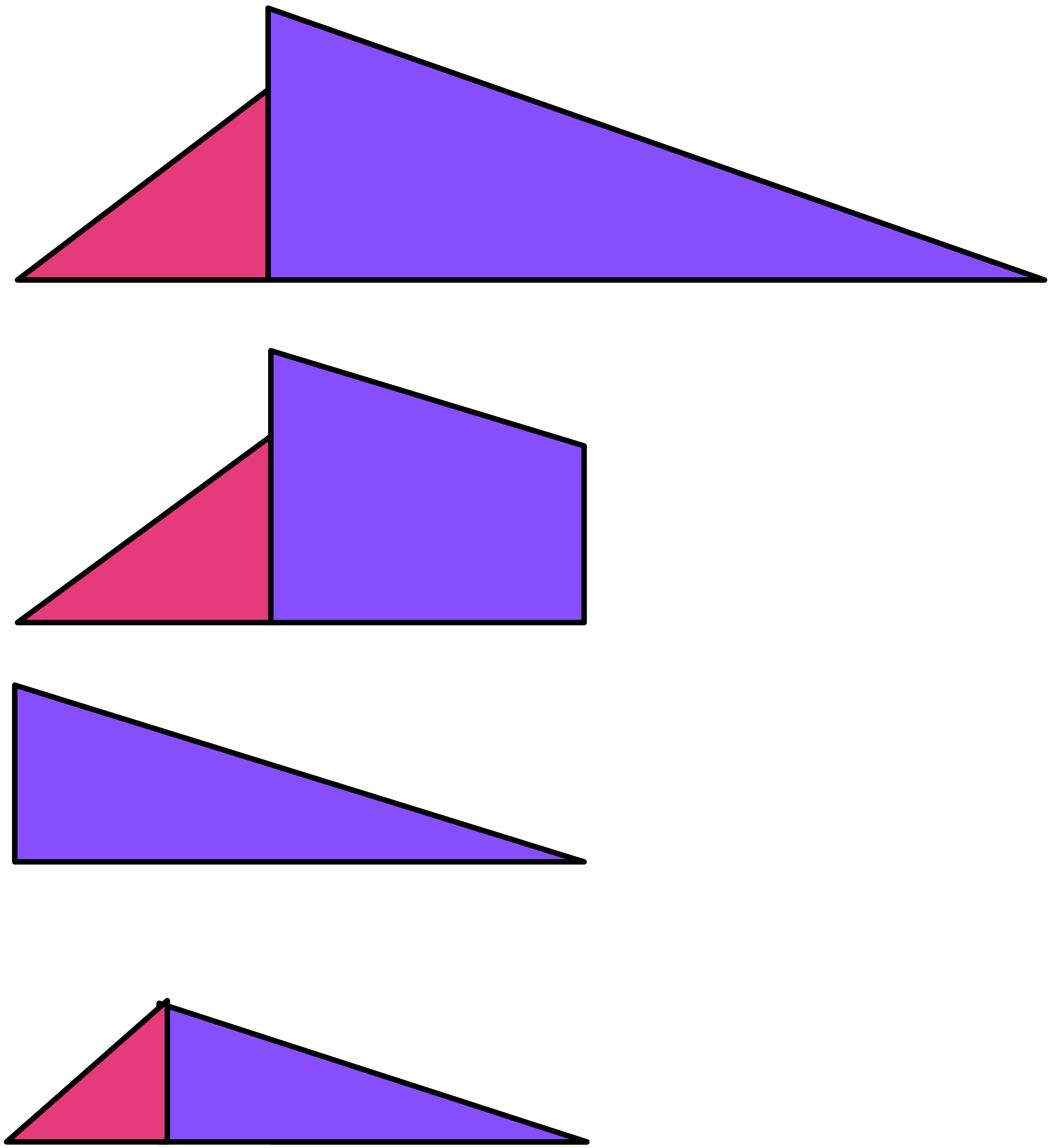


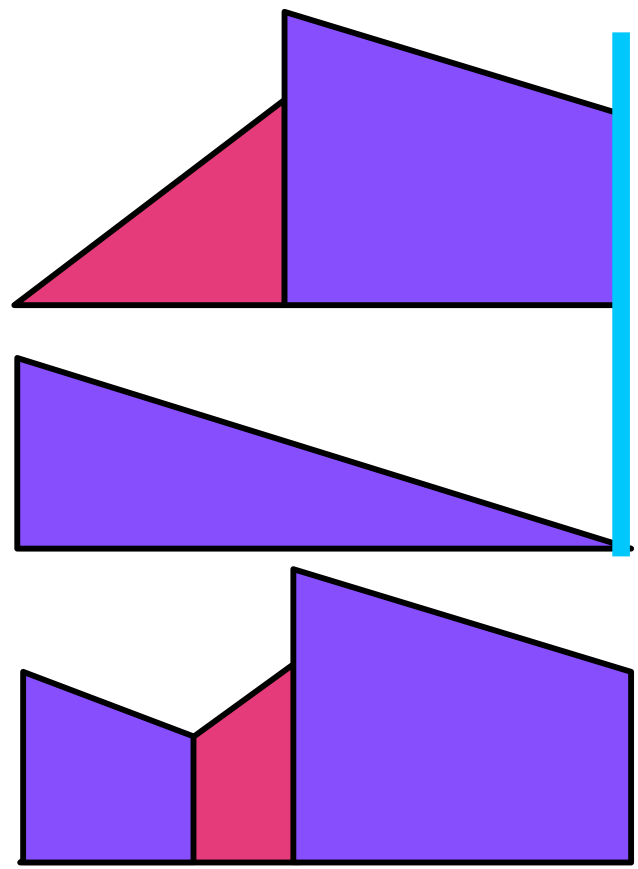
Splitting Bitonic Sequences - I

Sequence properties

s_1 and s_2 are both bitonic

$\forall_x \forall_y \; x \in s_1, y \in s_2, x < y$





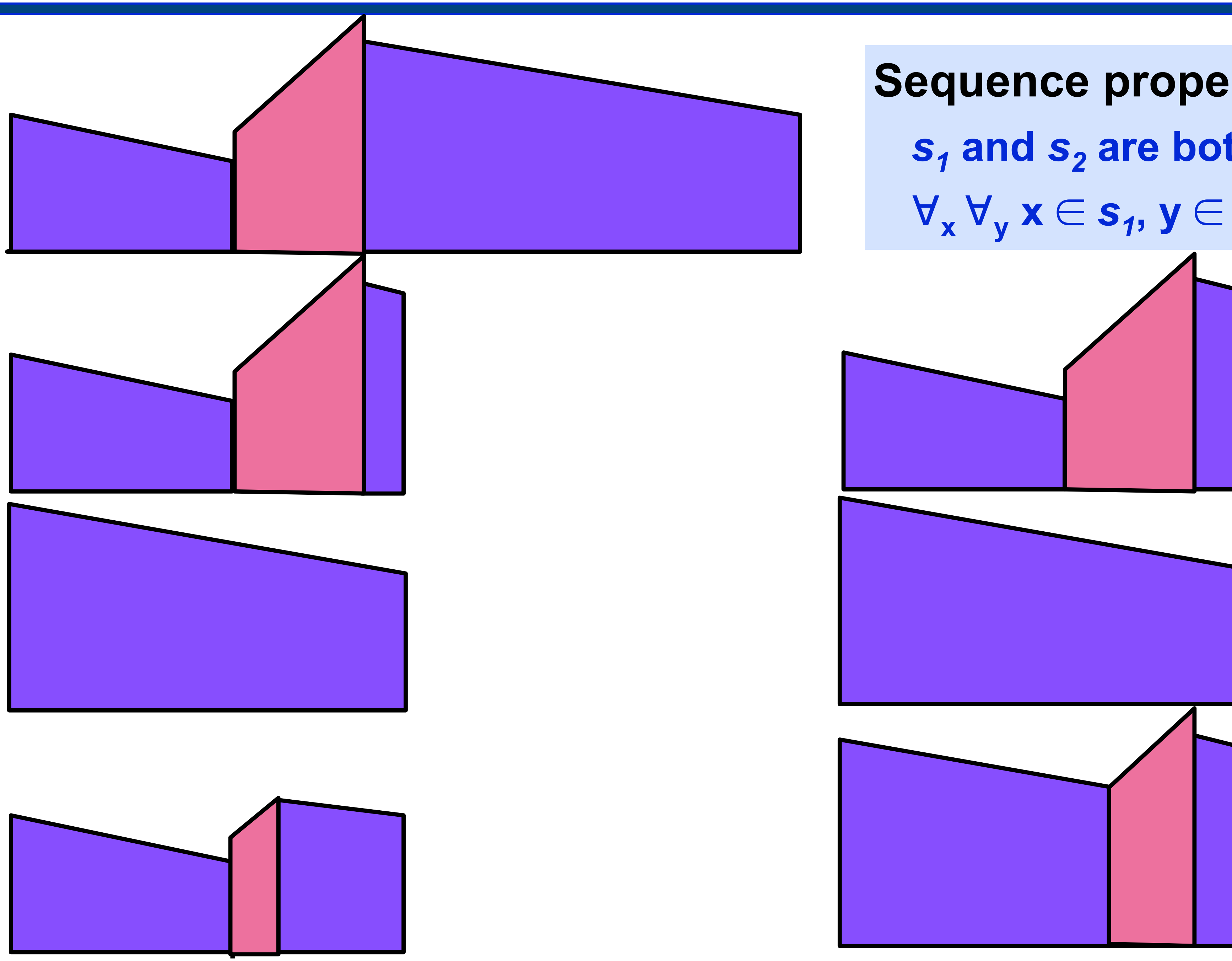
min

max

Splitting Bitonic Sequences - II

Sequence properties

- ***s_1* and *s_2* are both bitonic**
- **$\forall_x \forall_y \; x \in s_1, y \in s_2 \; , x < y$**



Bitonic Merge

Sort a bitonic sequence through a series of bitonic splits

Example: use bitonic merge to sort 16-element bitonic sequence

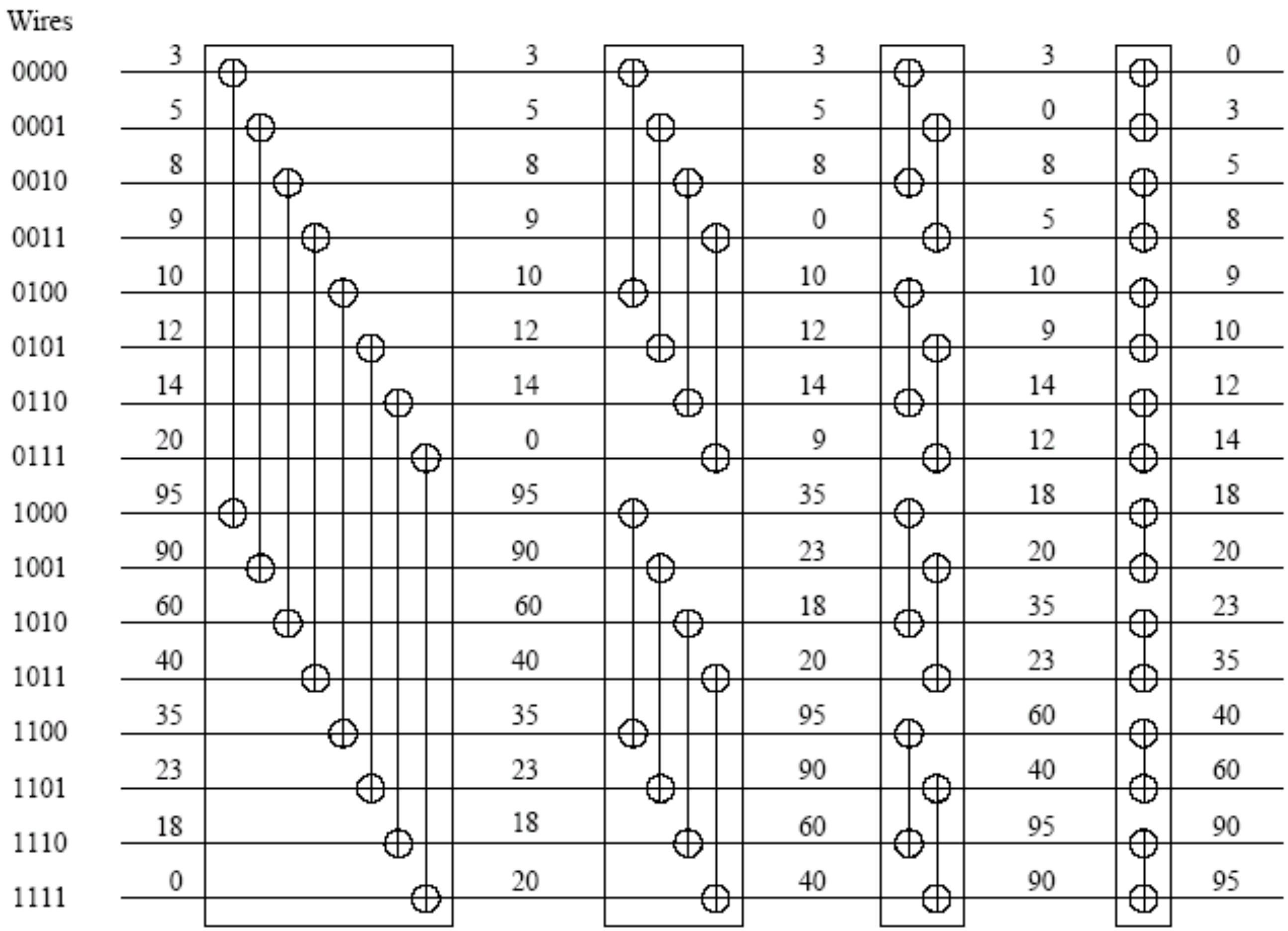
How: perform a series of $\log_2 16 = 4$ bitonic splits

Original sequence	3	5	8	9	10	12	14	20	95	90	60	40	35	23	18	0
1st Split	3	5	8	9	10	12	14	0	95	90	60	40	35	23	18	20
2nd Split	3	5	8	0	10	12	14	9	35	23	18	20	95	90	60	40
3rd Split	3	0	8	5	10	9	14	12	18	20	35	23	60	40	95	90
4th Split	0	3	5	8	9	10	12	14	18	20	23	35	40	60	90	95

Sorting via Bitonic Merging Network

- **Sorting network can implement bitonic merge algorithm**
  —*bitonic merging network*
- **Network structure**
  —$\log_2 n$ columns
  —each column
    – *n*/2 comparators
    – performs one step of the bitonic merge
- **Bitonic merging network with *n* inputs: ⊕BM[n]**
  —yields increasing output sequence
- **Replacing ⊕ comparators by ⊖ comparators: ⊖BM[n]**
  —yields decreasing output sequence

Bitonic Merging Network, ⊕ BM[16]



- **Input: bitonic sequence**
  - — input wires are numbered 0,1,…, *n* - 1 (shown in binary)
- **Output: sequence in sorted order**
- **Each column of comparators is drawn separately**

Bitonic Sort

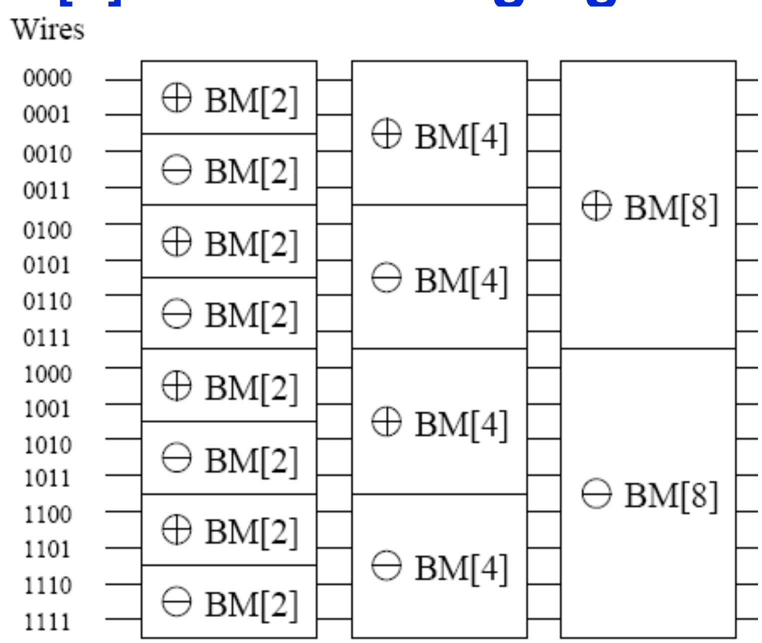
How do we sort an unsorted sequence using a bitonic merge?

Two steps

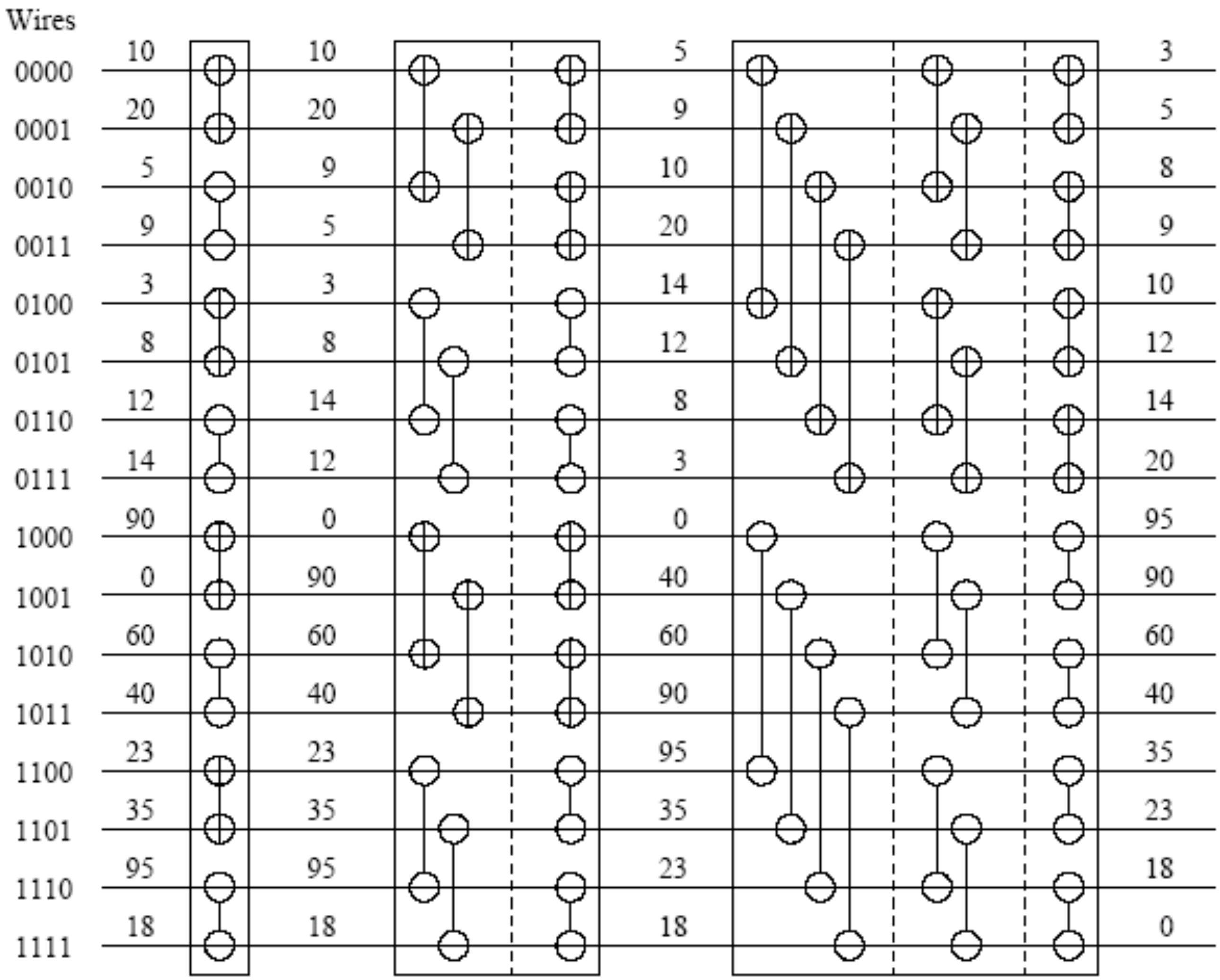
- **Build a bitonic sequence**
- **Sort it using a bitonic merging network**

Building a Bitonic Sequence

- **Build a single bitonic sequence from the given sequence**
  - —any sequence of length 2 is a bitonic sequence.
  - —build bitonic sequence of length 4
    - sort first two elements using ⊕BM[2]
    - sort next two using ⊖BM[2]
- **Repeatedly merge to generate larger bitonic sequences**
  - —⊕BM[*k*] & ⊖BM[*k*]: bitonic merging networks of size *k*



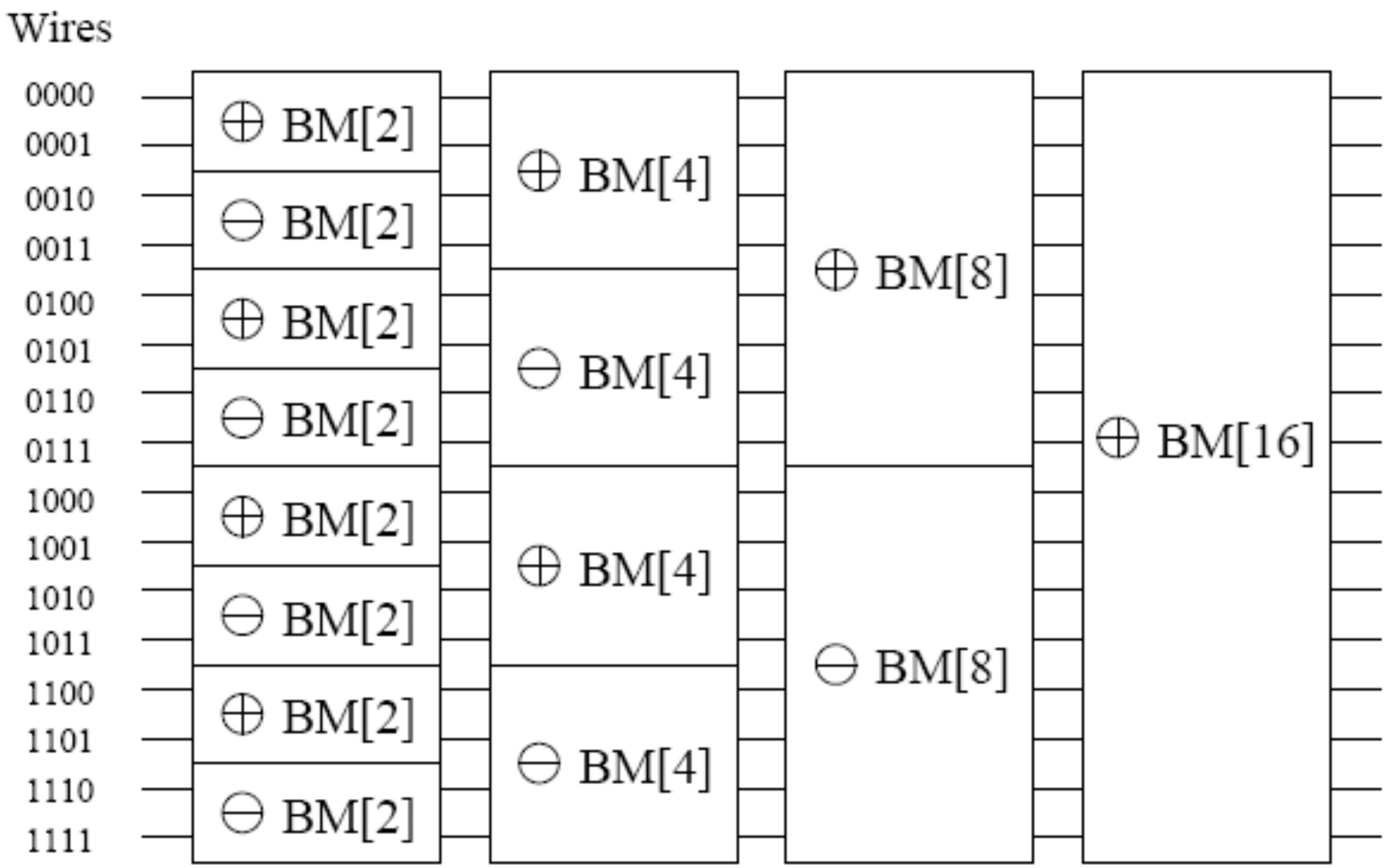
Building a Bitonic Sequence



Input: sequence of 16 unordered numbers

Output: a bitonic sequence of 16 numbers

Bitonic Sort, n = 16



- First 3 stages create bitonic sequence input to stage 4
- Last stage (⊕BM[16]) yields sorted sequence

Complexity of Bitonic Sorting Networks

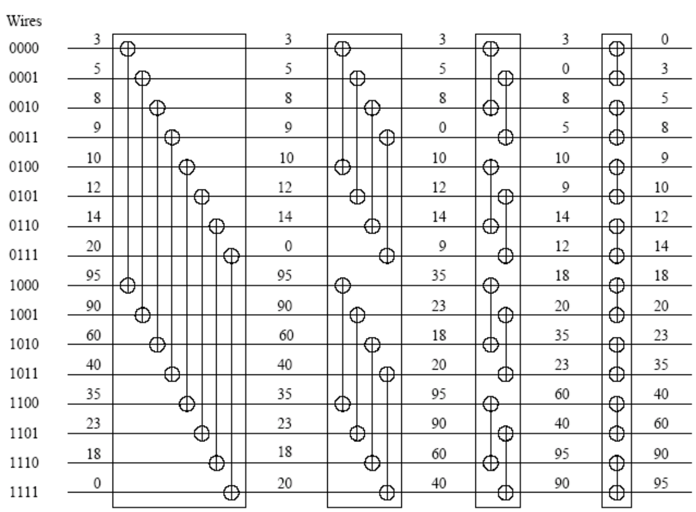
- **Depth of the network is** $\Theta(\log^2 n)$
  - **—$\log_2$ n merge stages**
  - **—j^{th} merge stage is $\log_2 2^j = j$**
  - **—depth =** $$\sum_{j=1}^{\log_2 n} \log_2 2^j = \sum_{i=1}^{\log_2 n} j = (\log_2 n + 1)(\log_2 n)/2 = \theta(\log^2 n)$$
- **Each stage of the network contains *n*/2 comparators**
- **Complexity of serial implementation =** $\Theta(n \log^2 n)$

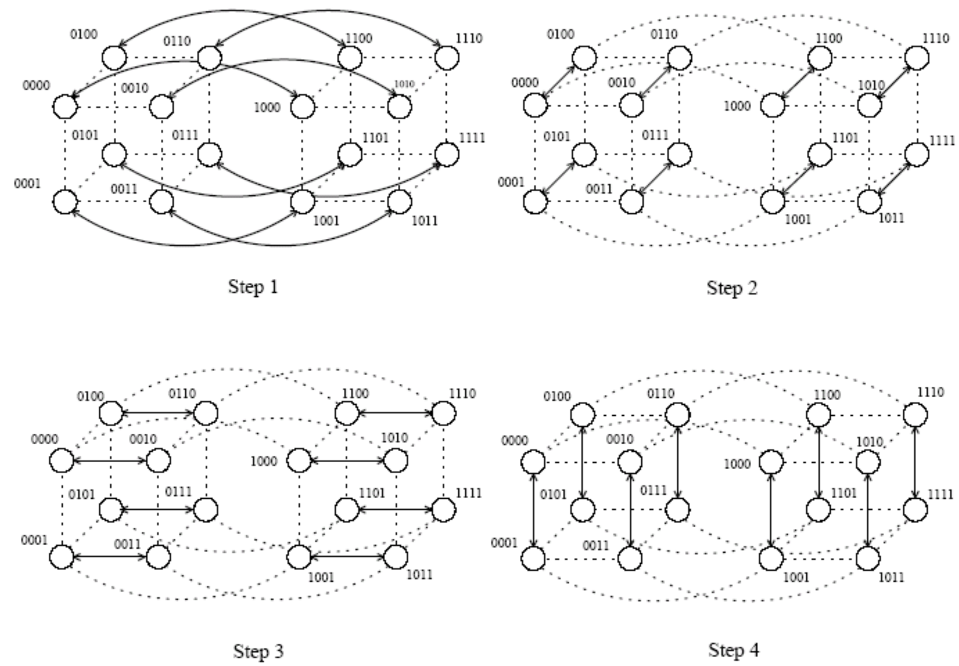
Mapping Bitonic Sort to a Hypercube

Consider one item per processor

- **How do we map wires in bitonic network onto a hypercube?**
- **In earlier examples**
  - **—compare-exchange between two wires when labels differ in 1 bit**
- **Direct mapping of wires to processors**
  - **—all communication is nearest neighbor**

Mapping Bitonic Merge to a Hypercube

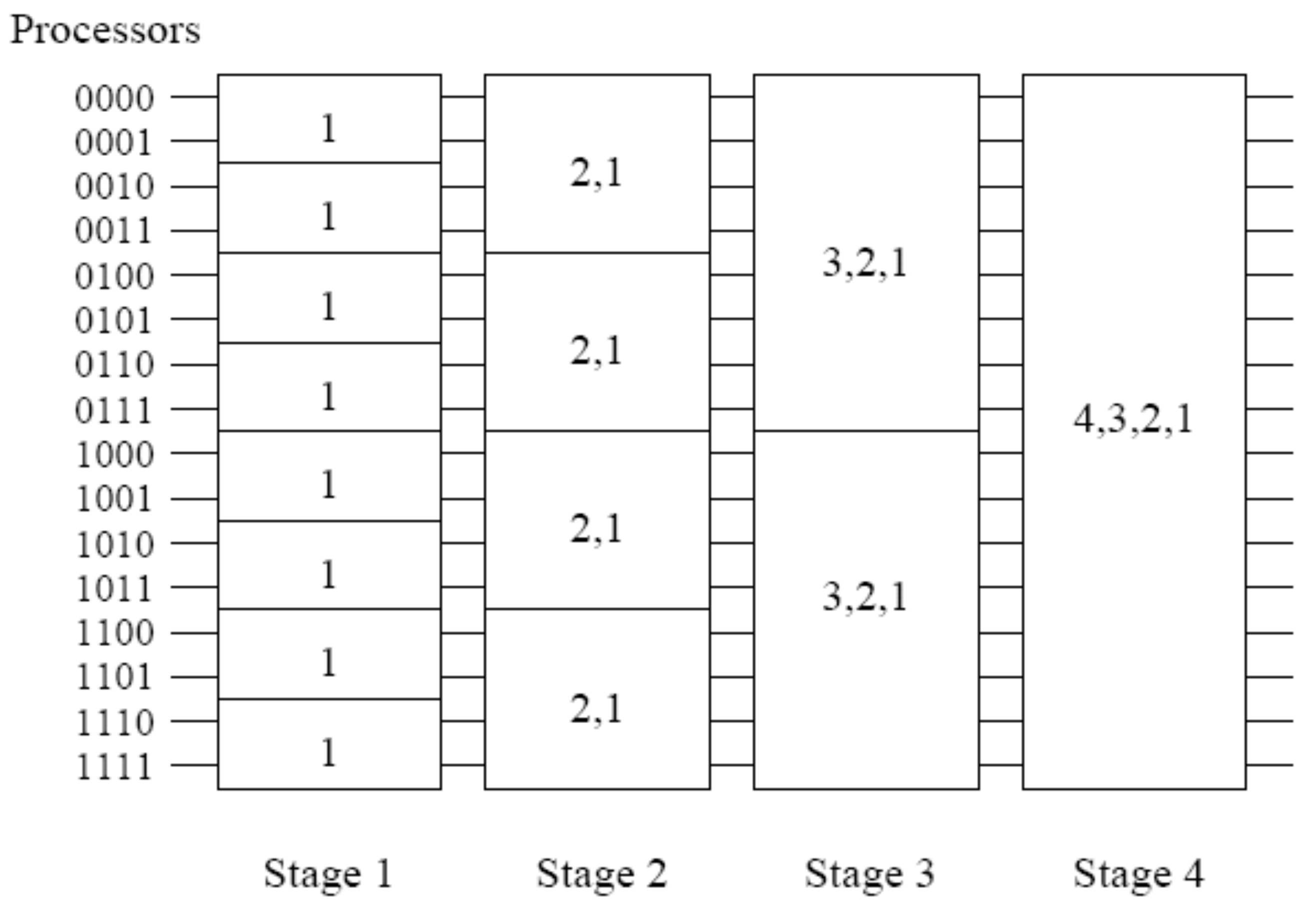




Communication during the last merge stage of bitonic sort

- **Each number is mapped to a hypercube node**
- **Each connection represents a compare-exchange**

Mapping Bitonic Sort to Hypercubes



Communication in bitonic sort on a hypercube

- **Processes communicate along dims shown in each stage**
- **Algorithm is cost optimal w.r.t. its serial counterpart**
- **Not cost optimal w.r.t. the best sorting algorithm**

Batcher's Bitonic Sort in NESL

```
function bitonic_merge(a) =
    if (#a == 1) then a
    else
        let
          halves = bottop(a)
          mins = {min(x, y) : x in halves[0]; y in halves[1]};
          maxs = {max(x, y) : x in halves[0]; y in halves[1]};
        in flatten({bitonic_merge(x) : x in [mins,maxs]});

    function bitonic_sort(a) =
    if (#a == 1) then a
    else
        let b = {bitonic_sort(x) : x in bottop(a)};
        in bitonic_merge(b[0]++reverse(b[1]));
```

Run this code at http://www.cs.rice.edu/~johnmc/nesl.html

Sample Sort

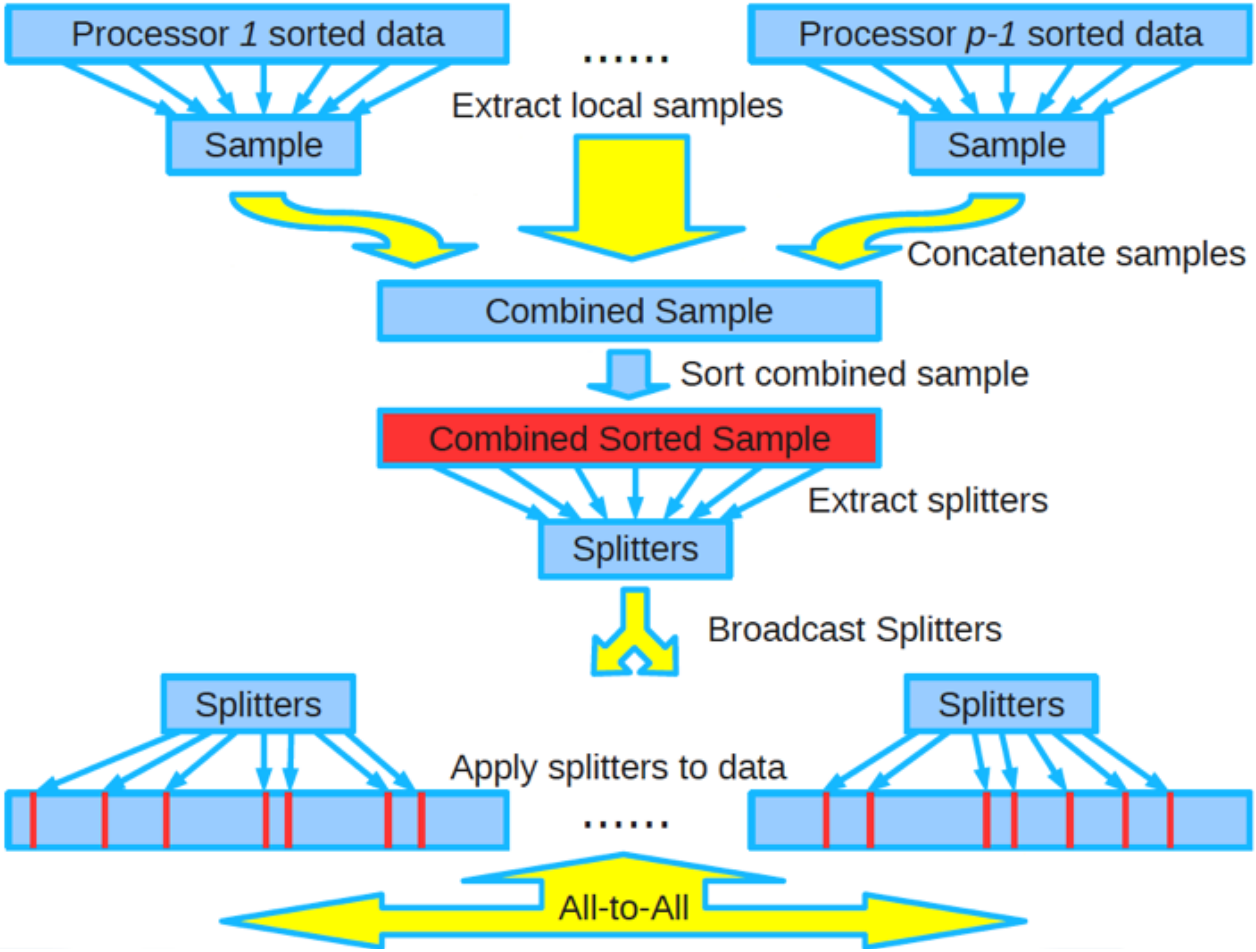


Figure Credit: Kale and Solmonik, IPDPS 2010

Sample Sort

- **Algorithm**
  - —each processor sorts its local data.
  - —each processor selects a sample vector of size p−1 from its local data. the k^{th} element of the vector is element n/p((k+1)/p) of local data.
  - —send samples to P_0. merge them there and produce a combined sorted sample of size p(p − 1).
  - —P_0 defines and broadcasts a vector of p−1 splitters with the k^{th} splitter as element p(k + 1/2) of the combined sorted sample.
  - —each processor sends its local data to the appropriate destination processors, as defined by the splitters, in one round of all-to-all communication.
  - —each processor merges the data chunks that it receives.
- **Notes [Shi, Shaeffer; JPDC 14:4, April 1992]**
  - —asymptotically optimal for $n \geq p^3$
  - —for n sufficiently large, no processor ends up with more than 2n/p keys
  - —scaling eventually limited by $O(p^2)$ sort of combined samples

Histogram Sort

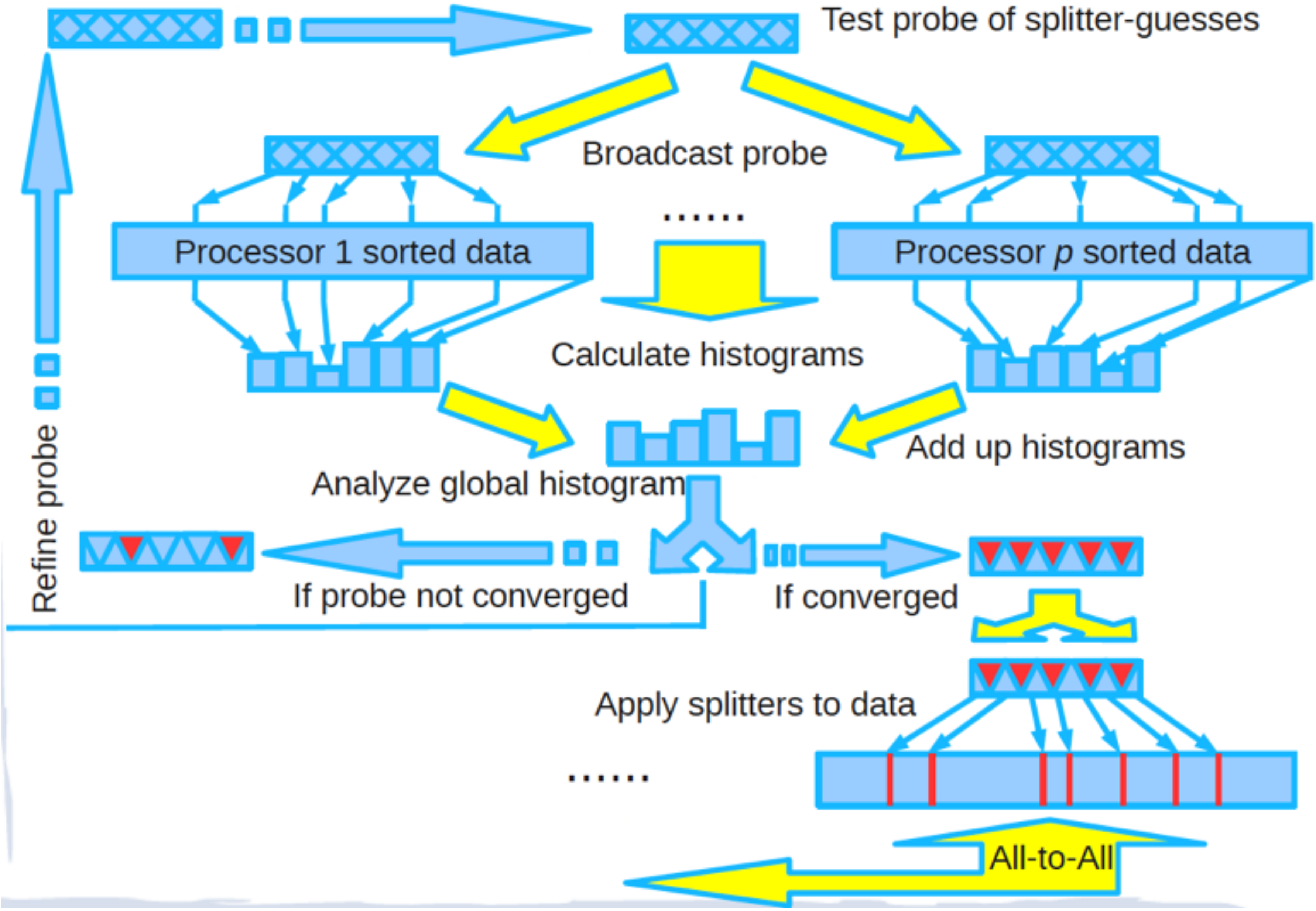


Figure Credit: Kale and Solmonik, IPDPS 2010

Histogram Sort

- Goal: divide keys into p evenly sized pieces
  —use an iterative approach to do so
- Initiating processor broadcasts k > p-1 splitter guesses
- Each processor determines how many keys fall in each bin
- Sum histogram with global reduction
- One processor examines guesses to see which are satisfactory
- Iterate if guesses are unsatisfactory
- Broadcast finalized splitters and number of keys for each processor
- Each processor sends local data to appropriate processors using all-to-all communication
- Each processor merges chunks it receives
- Kale and Solomonik improved this (IPDPS 2010)

Radix Sort

- In a series of rounds, sort elements into buckets by digit
  - —a k-bit radix sort looks at k bits every iteration
- Start with k least significant bits first, partition data into 2^k buckets
- Use an all-to-all pattern to distribute the buckets among the processors
- Each processor merges the buckets it receives
- Repeat until all bits have been considered
- O(bn/p) where b is the number of bits in a key
- Note: works best on a power of 2 number of processors
  - —even distribution of the 2^k buckets among the processors

Parallel Sorting Using Exact Splitters

Assumptions

- **Assumptions**
  - —distributed memory machines are ubiquitous
  - —cost of communication >> cost of computation
  - —large number of processors
  - —size of data >> number of processors
- **Design goal**
  - —move minimal amount of data over network

Then and Now

- CM-2 results from the 90s
  - —sample-based sort and radix sort are good in practice [Blelloch]
- Today
  - —cost of sampling is often quite high and sample sort requires redistribution at end
  - —sampling process requires well-chosen parameters to yield good samples
  - —can eliminate both steps if exact splitters can be determined quickly

Summary

- **Key idea**
  - find p-1 exact splitters in O(p log n) rounds of communication
- **Result**
  - close to optimal in computation and communication
    - moves less data than sample sorting, which is widely used
    - computationally a lot more efficient on distributed memory systems

Parallel Sorting with Exact Splitters

Algorithm.
Input: A vector v of n total elements, evenly distributed among p processors.
Output: An evenly distributed vector w with the same distribution as v, containing the sorted elements of v.

Notation
$d_i = |v_i|$
r_i = i^{th} global splitter

1. Sort the local elements v_i into a vector v'_i.
2. Determine the exact splitting of the local data:
   (a) Compute the partial sums $r_0 = 0$ and $r_j = \sum_{k=1}^{j} d_k$ for $j = 1 \ldots p$.
   (b) Use a parallel select algorithm to find the elements $e_1, \ldots, e_{p-1}$ of global rank $r_1, \ldots, r_{p-1}$, respectively.
   (c) For each r_j, have processor i compute the local index s_{ij} so that $r_j = \sum_{i=1}^{p} s_{ij}$ and the first s_{ij} elements of v'_i are no larger than e_j.
3. Reroute the sorted elements in v'_i according to the indices s_{ij}: processor i sends elements in the range $s_{ij-1} \ldots s_{ij}$ to processor j.
4. Locally merge the p sorted sub-vectors into the output w_i.

Local Sort

- **On each processor, sort the local data v_i into v'_i**
- **For a comparison-based sort, time = $O(\lceil n/p \rceil \lg \lceil n/p \rceil)$**

Selecting P-1 Exact Splitters

- Base case: single splitter selection
  —find a single splitter at global rank r
- Apply this algorithm p times (with like phases combined) to each of the desired splitters

Single Splitter Selection

- **First, consider first the problem of selecting one element with global rank r**
  - —elements may not be unique: want element whose set of ranks *contains* r
- **Define an active region on each P_i,**
  - —active range contains all elements that may still have rank r
  - —let a_i be its size
  - —initially, active range on each processor is v'_i
- **In each round, a pivot is found that partitions the active range in two. If the pivot isn't the target element, iterate on one of the partitions**

Single Splitter Selection

- Let each P_i compute m_i, the median of the active range of v'_i
- Use all-to-all broadcast to distribute all m_i
- Weight each median m_i by $a_i/(a_1 + a_2 + ... + a_p)$
  - —by definition, weights of medians $\{m_i \mid m_i < m_m\}$ sum to $\leq 1/2$
- Compute median of medians, m_m, in linear time
  - —M. Blum, R. W. Floyd, V. Pratt, R. L. Rivest, and R. E. Tarjan. 1973. Time bounds for selection. *J. Comput. Syst. Sci.* 7(4):448-461, August 1973.
- Find m_m with binary search over v'_i to determine f_i and l_i it can be inserted into vector v'_i
- Use all-to-all broadcast to distribute all f_i and l_i
- Compute $f = f_1 + f_2 + ... + f_p$ and $l = l_1 + l_2 + ... + l_p$. median m_m has rank $[f,l]$ in v
- If r in $[f,l]$ done; m_m is target element otherwise truncate active range
- If $l < r$, bottom index of active range is l_i+1
- If $r < f$, decrease top index to f_i-1
- Loop on truncated active range

Splitting by m_m will eliminate at least 1/4 of elements

—n elements initially, O(lg n) iterations

Simultaneous Selection

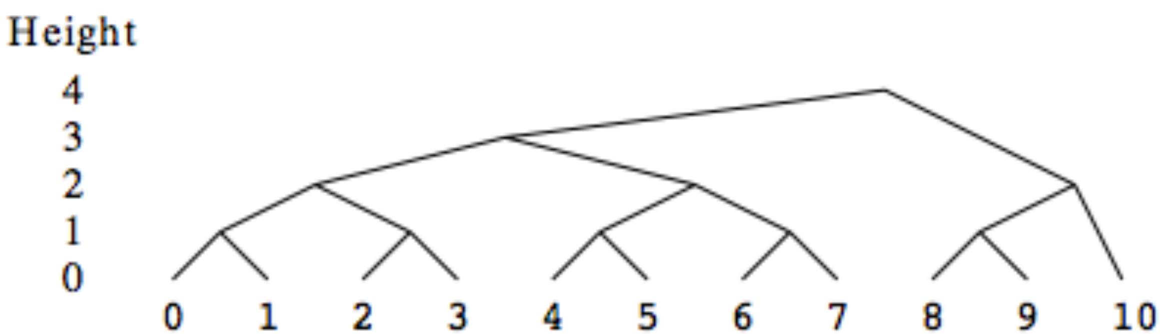
- Select multiple targets, each with different global rank
- For sorting, want p-1 elements of global rank
  - $d_1, d_1+d_2, \ldots, d_1+d_2+ \ldots + d_{p-1}$
- Simple strategy: call single selection for each desired rank
  - would increase communication rounds by O(p)
- Avoid this inflation by solving multiple selection problems independently, but combining their communication

Element Routing

- **Move elements from locations where they start to where they belong in sorted order**
- **Optimal parallel sorting algorithm: communicate every element from current location to a location in the remote array at most once**

Merging

- **Each processor has p sorted subvectors**
- **Must merge them into sorted sequence**
- **Approach**
  - —build a binary tree on top of the vectors
  - —for $P \neq 2^k$, a node of height i has at most 2^i leaf descendants

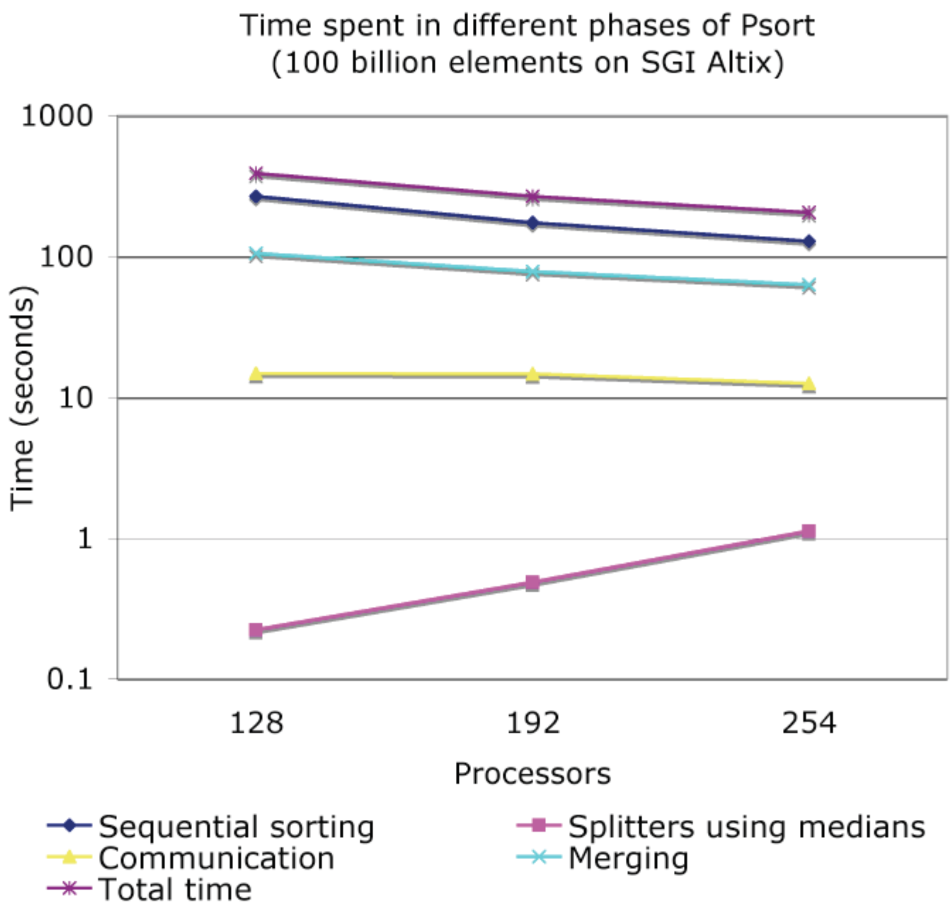


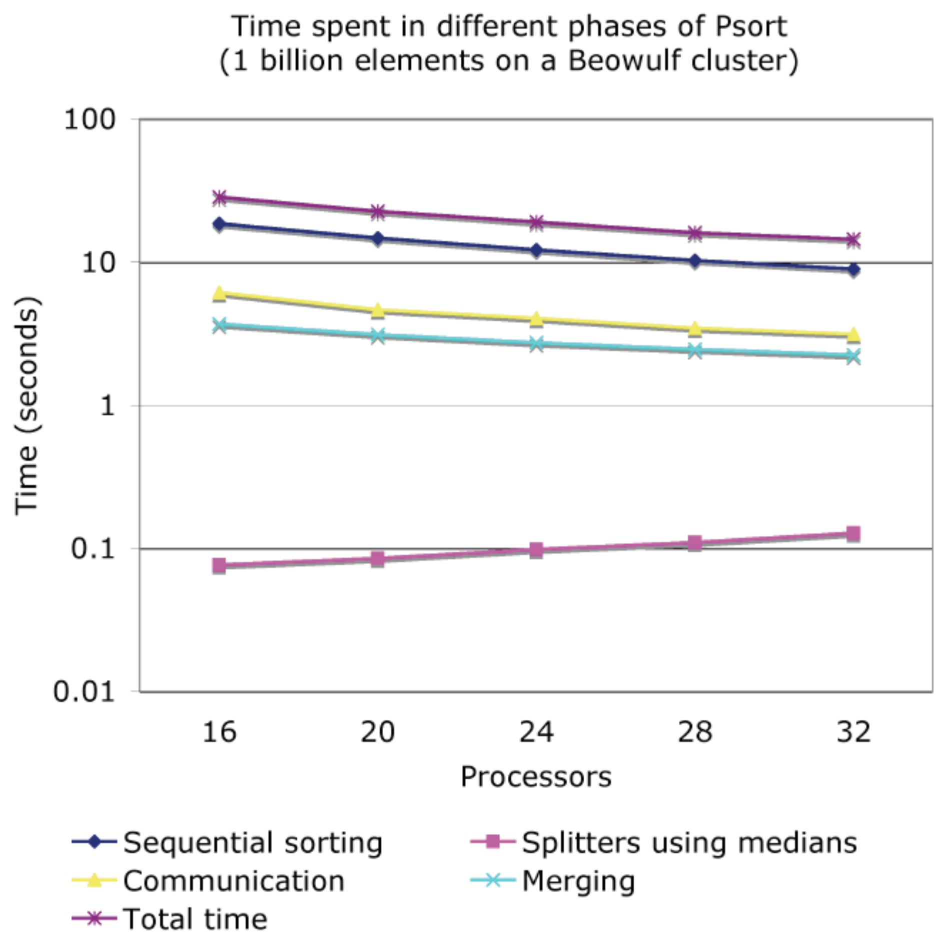
  - —tree has height $\lceil \lg p \rceil$
  - —merge pairs of subvectors guided by this tree
  - —each element moves at most $\lceil \lg p \rceil$ times
  - —total computation time on slowest processor $\lceil n/p \rceil \lceil \lg p \rceil$

Experimental Setup

- **Implementation**
  - C++ and MPI
  - used Standard Template Library std::sort and std::stable sort for sequential sort
- **Platforms**
  - SGI Altix
    - 256 Itanium 2 processors, 4TB RAM total
  - Beowulf cluster
    - 32 Xeon processors, 3GB of memory per node
    - Gigabit Ethernet interconnect

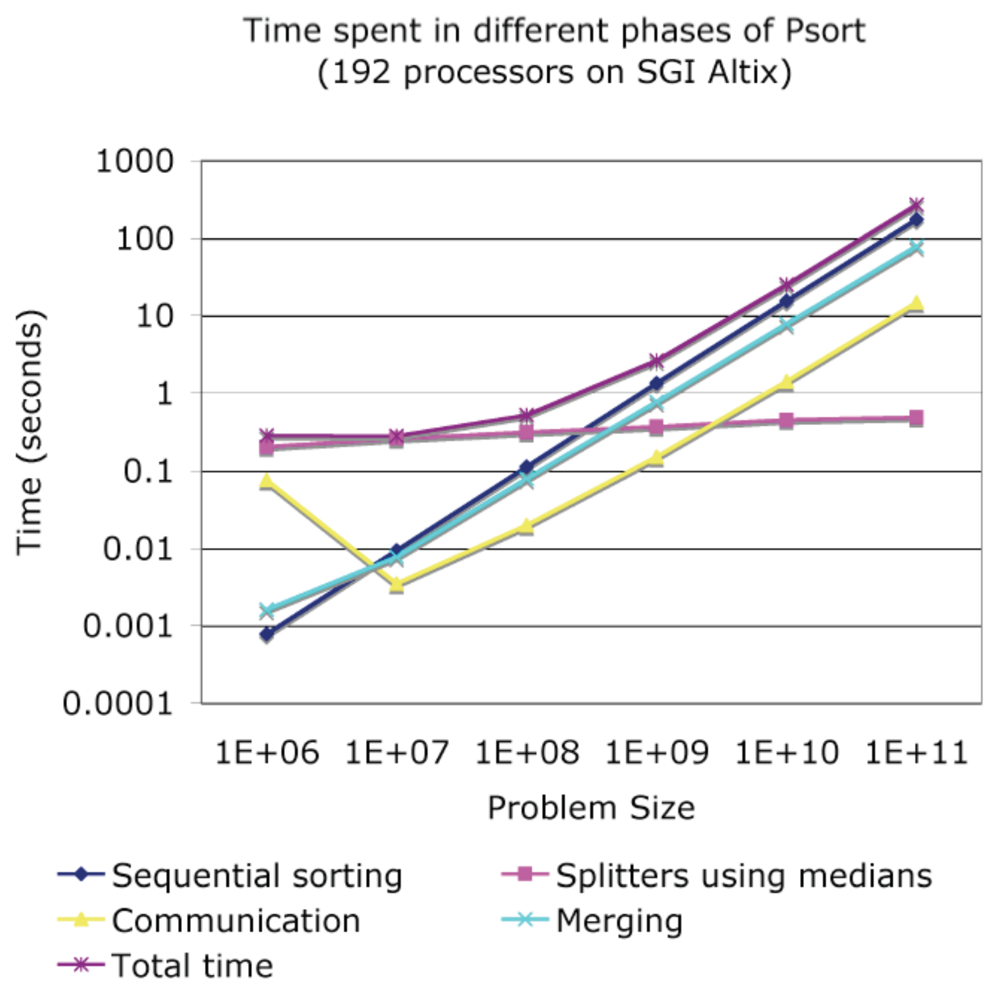
Time Spent in Different Phases, Scaling P

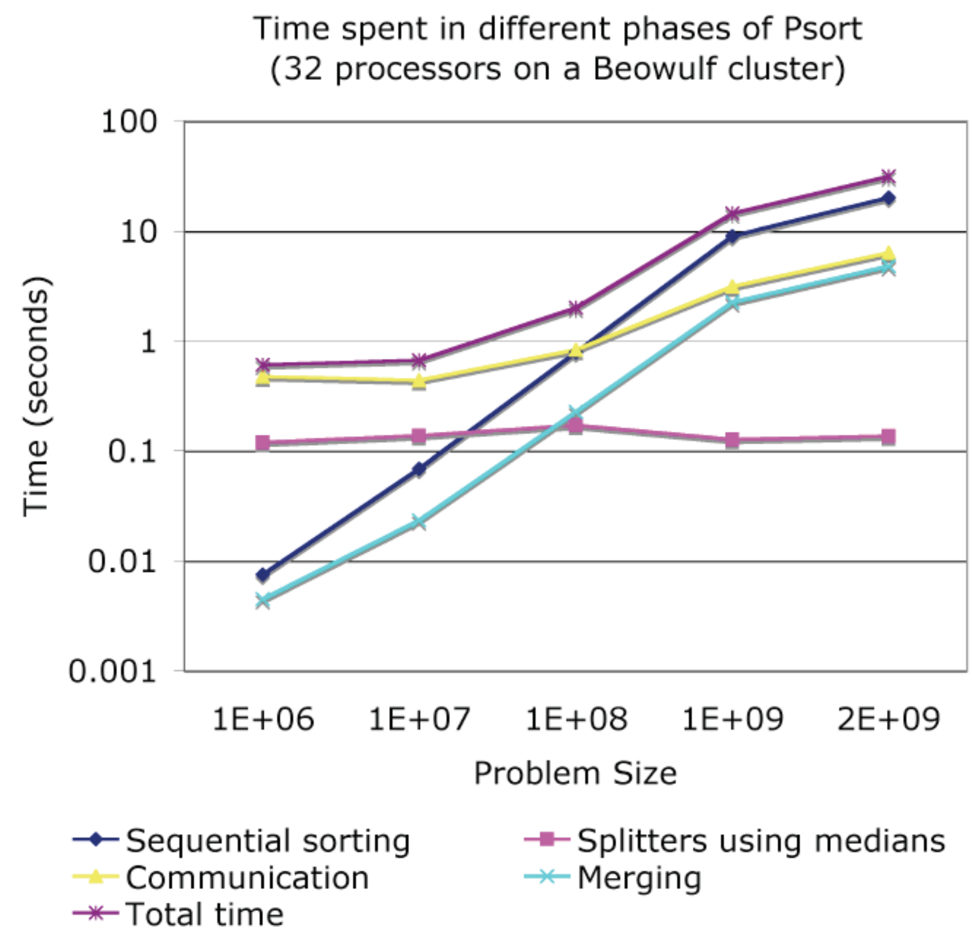




low and flat
is better

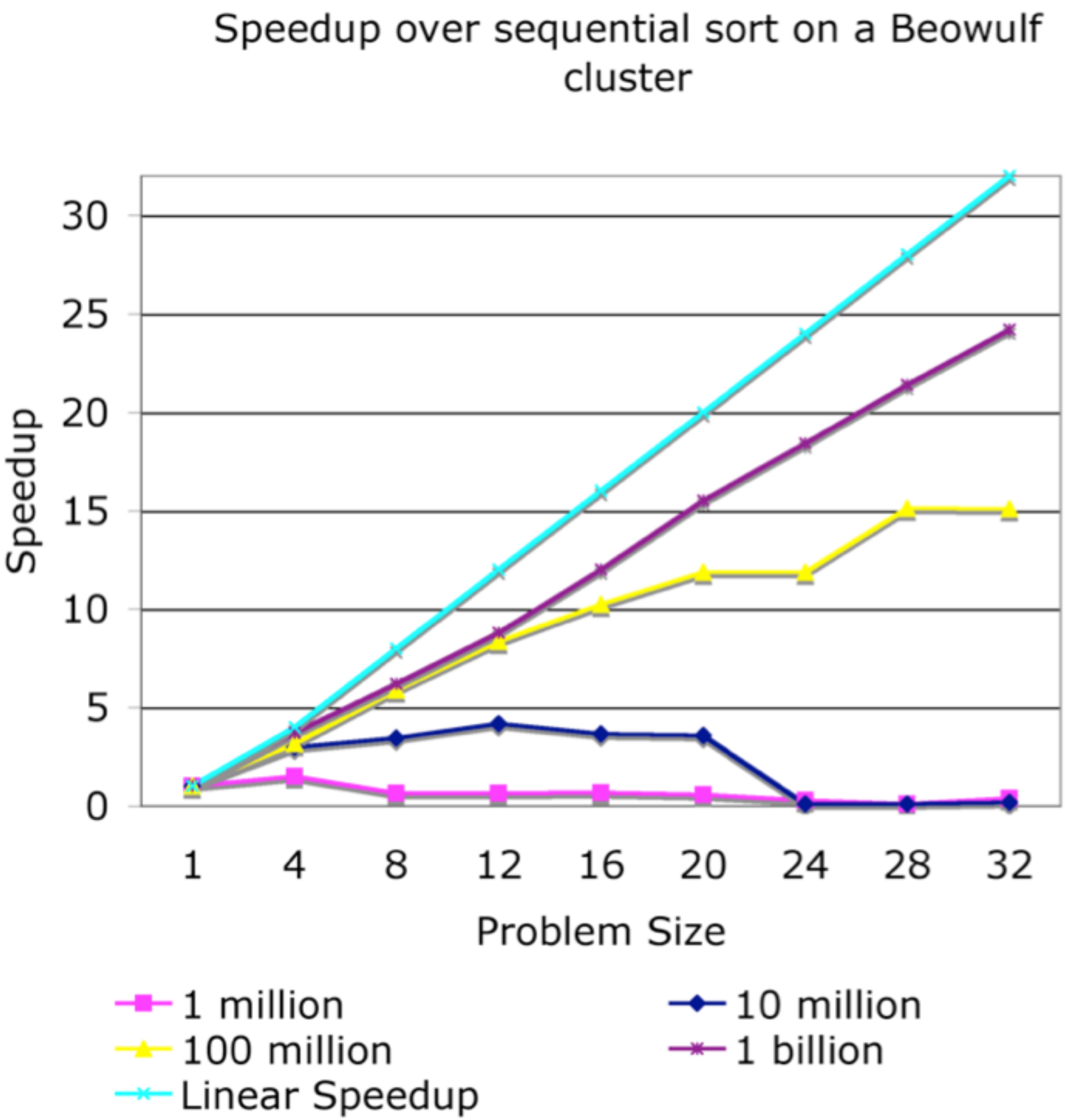
Time Spent in Different Phases, Scaling N





low and flat is better

Speedup vs. Data Size

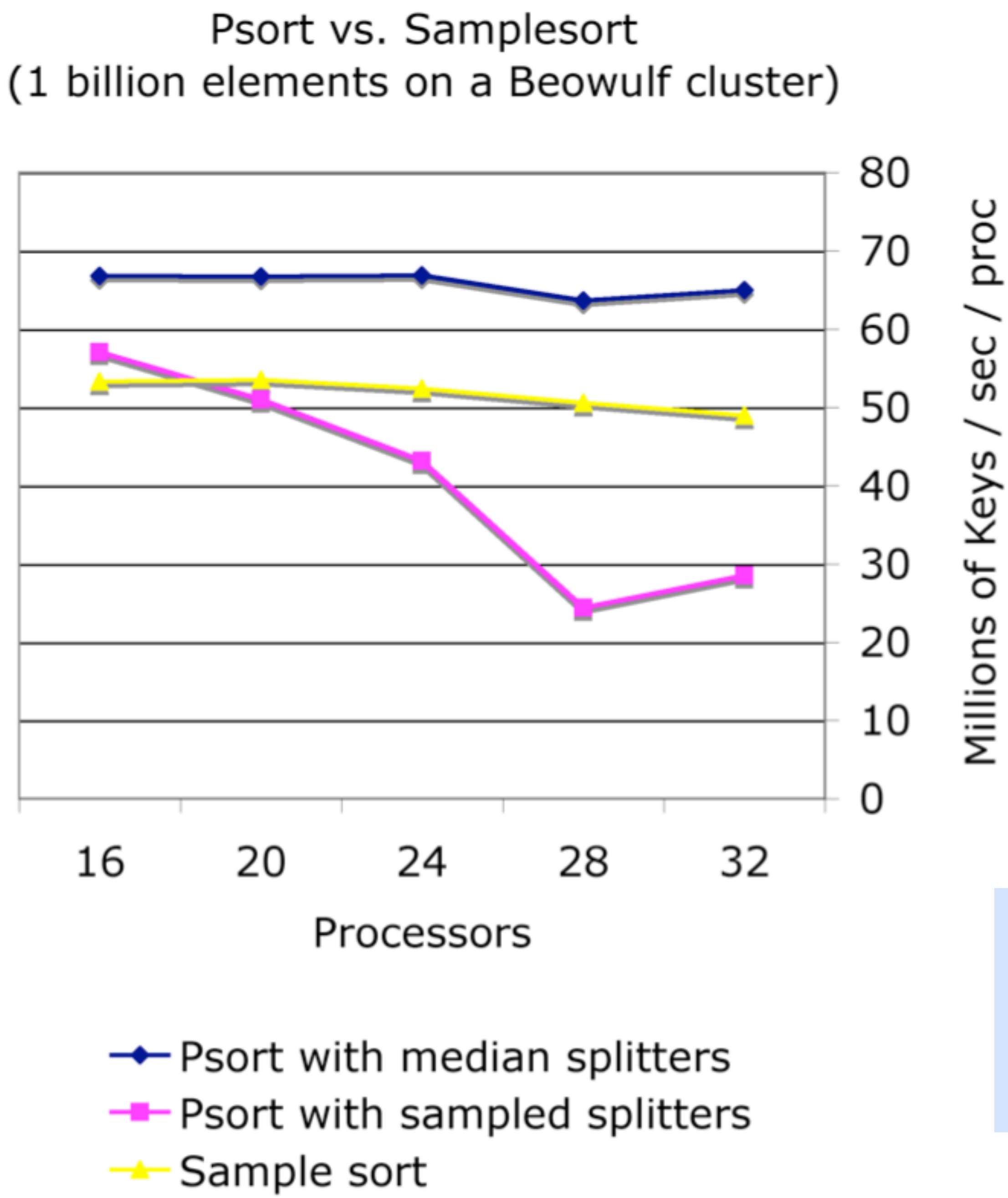


45 ° is best

Comparison with Sample Sort

- Psort
- Psort with sampled splitters
  - —same algorithm, but use random sampling to pick splitters instead of medians
- Sample sort
  - —traditional sampling based sorting algorithm, and based on the following steps:
    1. Pick splitters by sampling or oversampling.
    2. Partition local data to prepare for the communication phase.
    3. Route elements to their destinations.
    4. Sort local data.
    5. Redistribute to adjust processor boundaries.

Comparison with Sample Sort



high and flat is better

Things to Consider

- Distributed memory or shared memory
- Latency vs. bandwidth of communication
- Size of data vs. size of processors
- Asymptotic complexity of algorithm
  - —is P^2 too large

References

- Adapted from slides “Sorting” by Ananth Grama
- Based on Chapter 9 of “Introduction to Parallel Computing” by Ananth Grama, Anshul Gupta, George Karypis, and Vipin Kumar. Addison Wesley, 2003
- “Programming Parallel Algorithms.” Guy Blelloch. Communications of the ACM, volume 39, number 3, March 1996.
- http://www.cs.cmu.edu/~scandal/nesl/algorithms.html#sort
- Edgar Solomonik and Laxmikant V. Kale. Highly Scalable Parallel Sorting. Proceedings of IPDPS 2010.
- D. Cheng, V. Shah, J. Gilbert, A. Edelman. A Novel Parallel Sorting Algorithm for Contemporary Architectures. May 2007 http://gauss.cs.ucsb.edu/publication/psort.pdf