Parallel Sorting

Sequential Sorts

- Selection sort, Insertion sort, Bubble sort
  - $O(N^2)$
- Quicksort, Merge sort, Heap sort
  - $O(N \log N)$
  - Quicksort best on average
- Optimal PARALLEL time complexity
  - $O(N \log N) / P$
  - Note: if $P = N$, $O(\log N)$

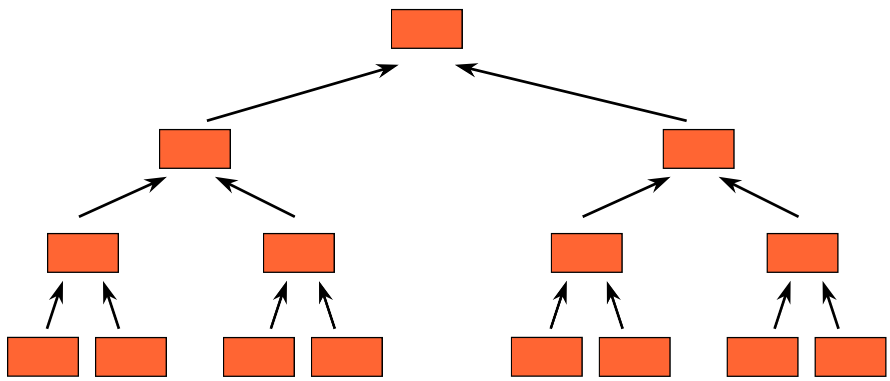
Designing a Parallel Sort

- Use a sequential sort and adapt
  - how well can it be done in parallel
  - sometimes a poor sequential algorithm can develop into a reasonable parallel algorithm (e.g. bubble sort -> odd-even transposition sort)
- Develop a different approach
  - harder, but sometimes leads to better solutions

Divide and Conquer approaches

- Merge sort
  - collects sorted list onto one processor, merging as items come together
  - maps well to tree structure, sorting locally on leaves, then merging up the tree
  - as items approach root of tree, processors drop out of merging process, limiting parallelism
  - O(log N), if P = N

Parallel Merge Sort Pattern



Divide and Conquer Approaches

- Quick sort
  - maps well to hypercube
  - divide list across dimensions of the hypercube, then sort locally
  - selection of partition values is even more critical than for sequential version since it affects load balancing
  - hypercube version leaves different numbers of items on different processors

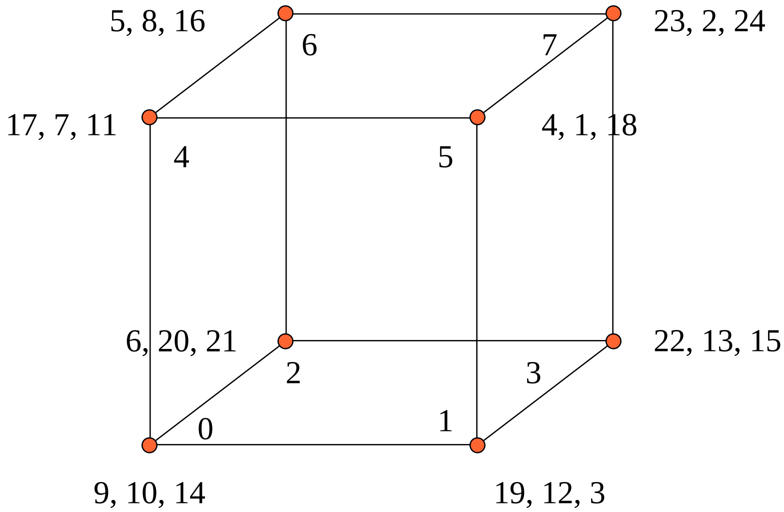
Quick Sort on Hypercube

1. Select global partition value, split values across highest cube dimension, high values going to upper side, low values to lower side (using gray code numbering of processing nodes)
2. Repeat on each of the one lower dimensional cubes forming the upper and lower halves of the original.
3. Continue this process until the remaining cube is a single processor, then sort locally.
4. Each node contains a sorted list, and the lists from node to node are in order (using processing node numbers.)
5. O(N/P log (N/P))

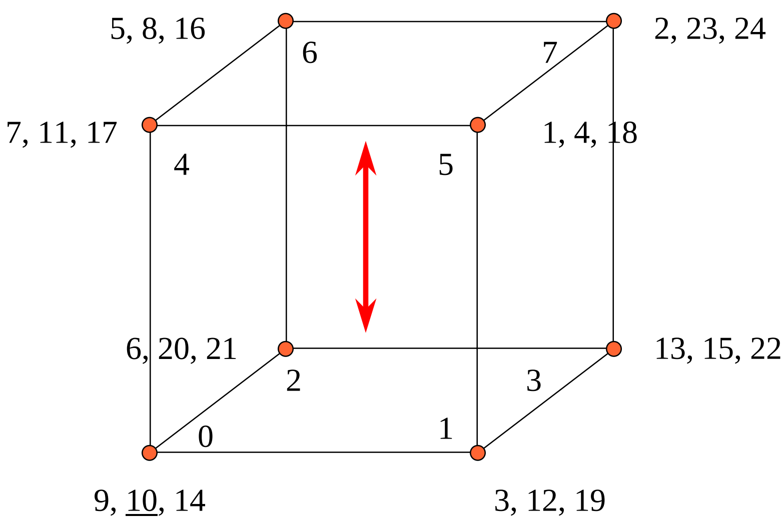
Divide and Conquer approaches

- Hyper Quicksort
  1. Divide data equally among nodes
  2. Sort locally on each node first
  3. Broadcast median value from node 0 as first pivot
  4. Each lists splits locally, then trades halves across highest dimension
  5. Apply steps 3 and 4 successively (and in parallel) to lower dimensional cube forming the two halves, and so on until dimension reaches 0

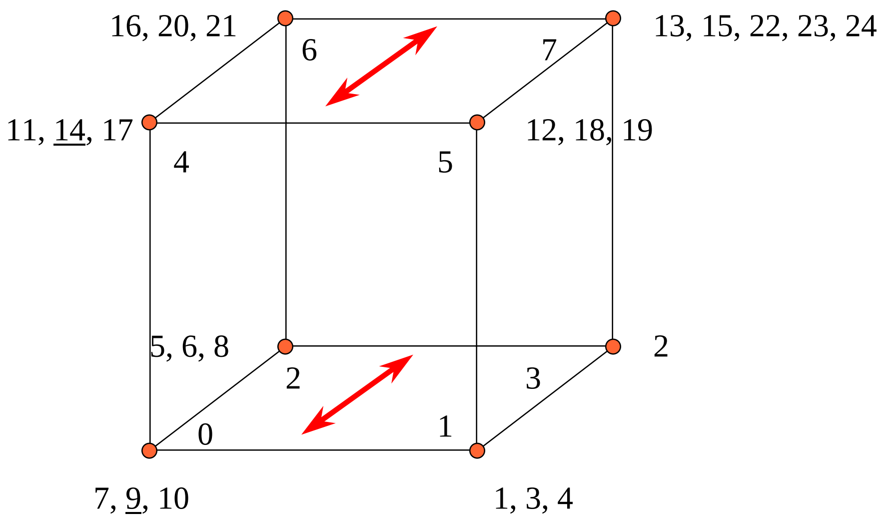
Hyper Quicksort, example



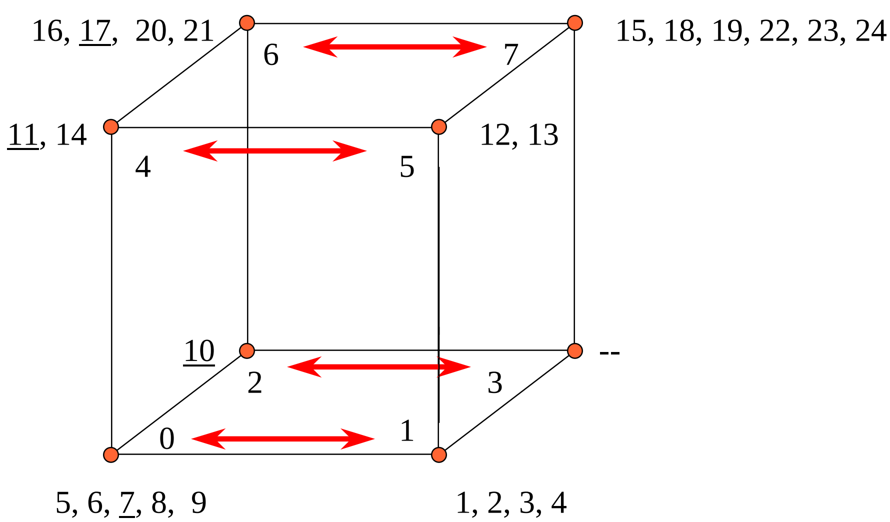
Sort locally, interchange vertically



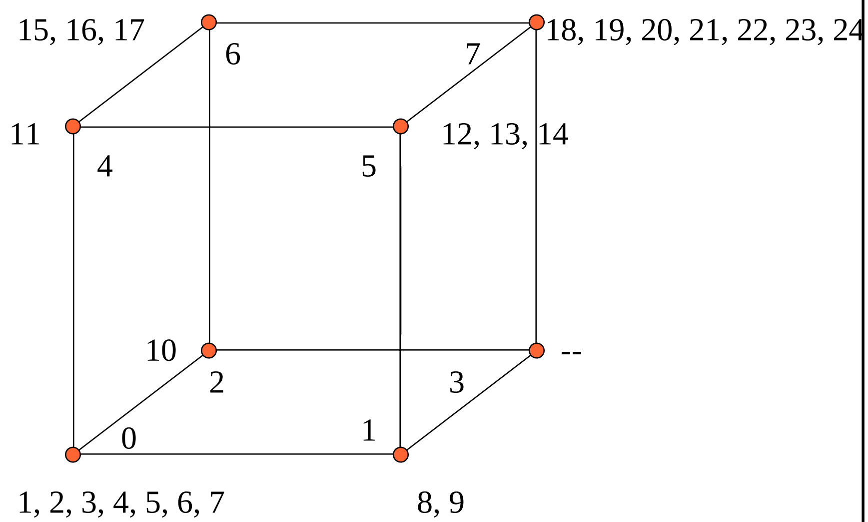
After first exchange, broadcast medians for second exchange



After second exchange, broadcast medians for third exchange



Completed sort



New Idea -- Bitonic Sort

- Developed for a sorting network
- Based on bitonic merge
  - start with bitonic sequence
  - finish with sorted (monotonic) sequence

Bitonic Sequence

$a_1, a_2, a_3, \ldots, a_n$ is bitonic if there is a k such that

$a_1 \le a_2 \le \ldots \le a_k \ge a_{k+1} \ge \ldots \ge a_n$ OR

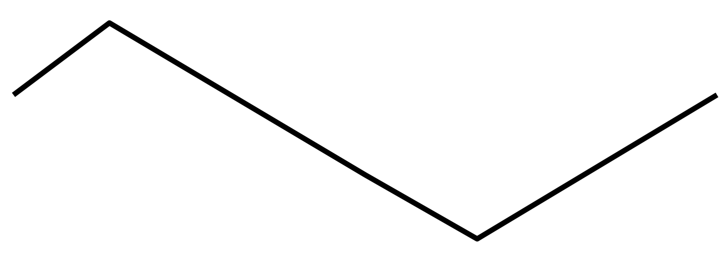
there is a cyclic shift of the sequence such that this is true.

Examples:

2, 4, 5, 7, 9, 8, 6, 3, 1

7, 9, 8, 6, 3, 1, 2, 4, 5

Bitonic sequence

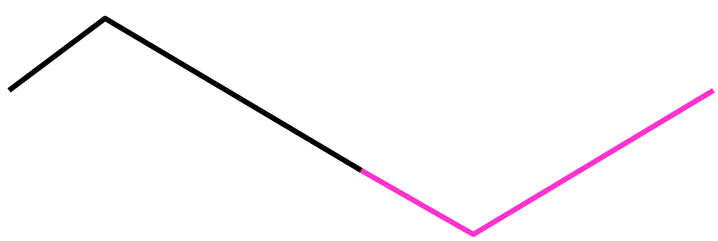


Split one bitonic into two

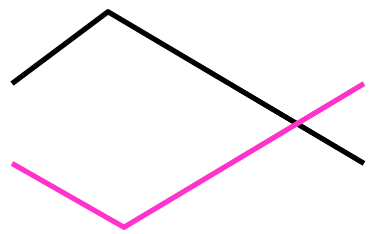
- Given $a_1, a_2, a_3, \ldots, a_{2n}$ is bitonic
- Let $c_k = \min(a_k, a_{n+k})$
- Let $d_k = \max(a_k, a_{n+k})$
- Then the sequences $c_1, c_2, \ldots c_n$ and $d_1, d_2, \ldots, d_n$ are both bitonic AND all the c_k are less than all the d_k

Graphical Proof

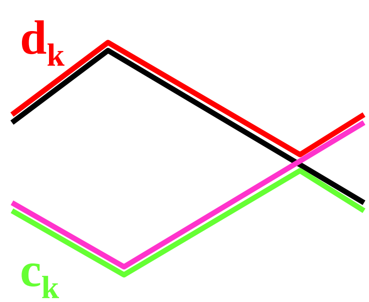
Bitonic sequence



Bitonic sequence, overlay halves



Bitonic sequence, max and min



Bitonic Merge

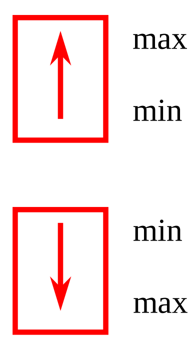
Given $a_1, a_2, \ldots, a_{2n}$, a bitonic sequence

1. Carry out pair-wise (a_k, a_{k+n}) min-max comparison and form sequences

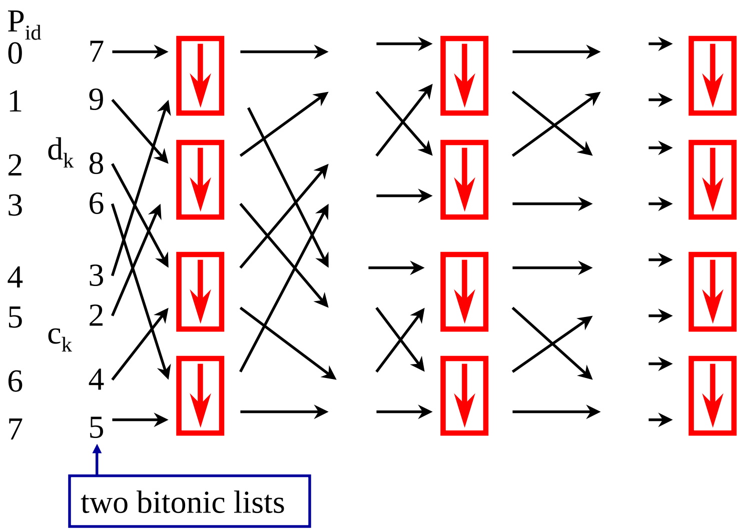
   $c_1, c_2, \ldots, c_n, d_1, d_2, \ldots, d_n$ so that all c_k <= all d_k and each half is bitonic.
2. Apply the same procedure recursively (or in parallel) to each half of the list, and so on, until each bitonic sublist consists of exactly one element
3. The list is now in increasing order.

Bitonic merge network

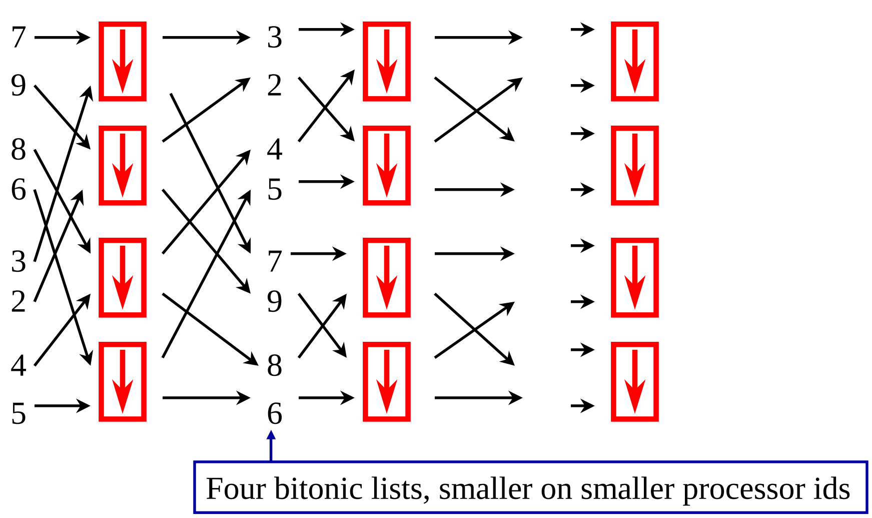
- Based on simple comparator
  - two inputs, two outputs
  - upper output is max of inputs
  - lower output is min of inputs
  - Reversed arrow indicates lower is max, upper is min



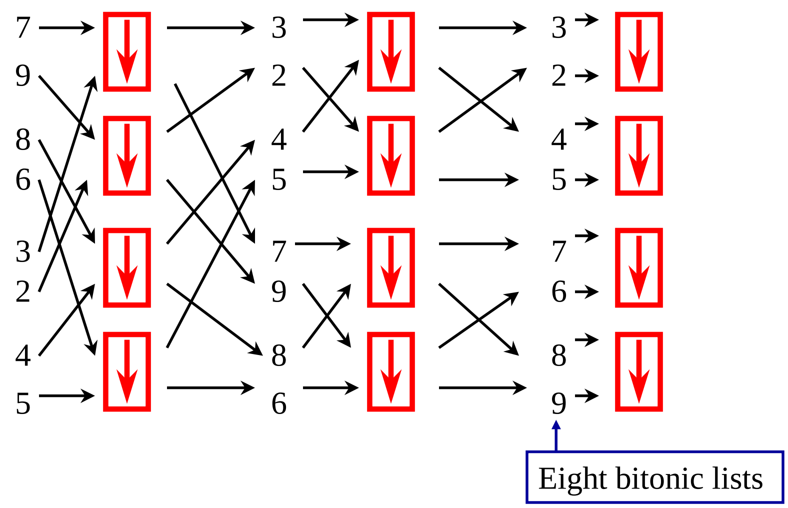
Bitonic merge network (8 element)



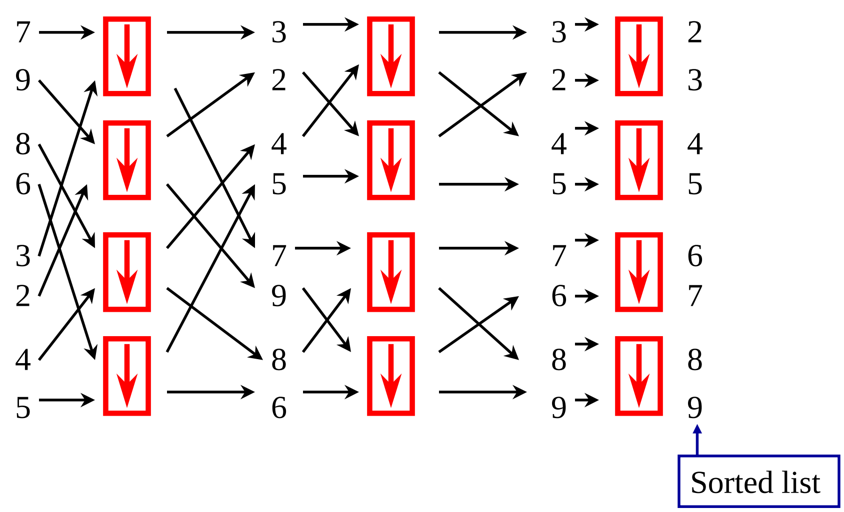
Bitonic merge network (8 element)



Bitonic merge network (8 element)



Bitonic merge network (8 element)



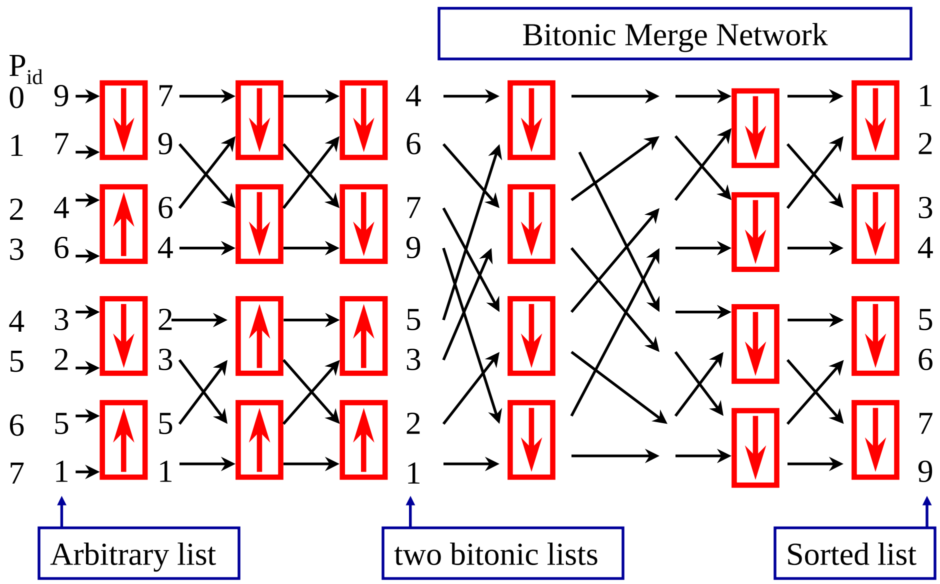
Bitonic Sort

- A series of bitonic merges
  1. Pairwise compare to create ordered pairs, alternating increasing and decreasing pairs.
  2. Sets of four now form bitonic sequences: carry out bitonic merge on fours, alternating order to create fours alternately in increasing and decreasing order.
  3. Sets of eight now form bitonic sequences: carry out bitonic merge on eights, alternating order to create eights alternately in increasing and decreasing order.

Bitonic Sort

- Continue this process, creating longer and longer bitonic sequences, until the whole sequence is bitonic and the final bitonic merge creates a sorted list.

Bitonic Sorting Network (8 element)



Bitonic Sort, Analysis

- Assume list length is $N = 2^k$
- The bitonic merge stages have 1, 2, 3, …, k steps each, so time to sort is

  time = 1 + 2 + … + k = k (k-1) / 2

  = $O(k^2) = O(\log^2 N)$
- Each step needs N/2 processors, so the total number of processors is $O((N/2)(\log^2 N))$
- Many lists can be pipelined, so lists can be output at the rate of one per time step.

Bitonic Sort, Hypercube

- Each input line corresponds to a node of the hypercube
- Each comparison box corresponds to an exchange between nodes along hypercube connections
- Larger lists divided among nodes, sorted locally, merge exchange between nodes according to bitonic pattern.

Compare & Exchange Algorithms

- Odd-Even Transposition Sort
- Shearsort

Odd-Even Transposition

- Based on bubble sort
- Compare even with next higher position
  - swap if needed
- Compare odd with next higher position
  - swap if needed
- How many iterations needed for N items?

15	2	3	8	0	11	9	14	12	1	5	4	7	6	10	13
2	15	3	8	0	11	9	14	1	12	4	5	6	7	10	13
2	3	15	0	8	9	11	1	14	4	12	5	6	7	10	13
2	3	0	15	8	9	1	11	4	14	5	12	6	7	10	13
2	0	3	8	15	1	9	4	11	5	14	6	12	7	10	13

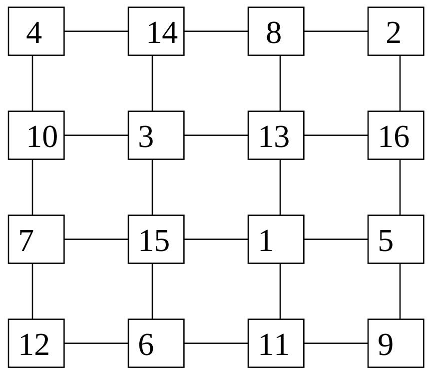
Odd-Even Transposition Sort VRML

- http://www.cs.rit.edu/~icss571/parallelwrl/oets.wrl

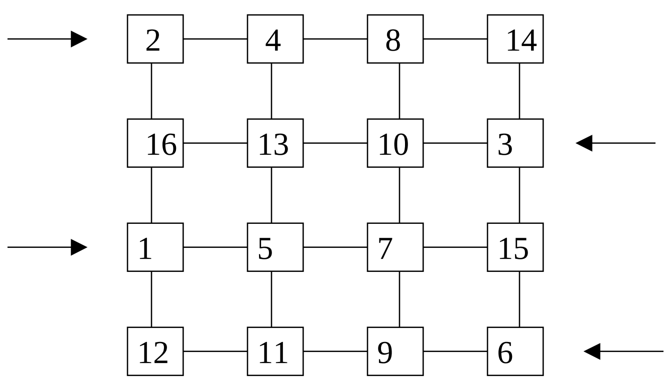
Shearsort

- 2D sorting: O(SQRT(N) (log N + 1))
- Odd phases: each row sorted independently in alternate directions
  - Even rows: smallest on left
  - Odd rows: smallest on right
- Even phases: Each column sorted independently top to bottom
- Log N + 1 iterations

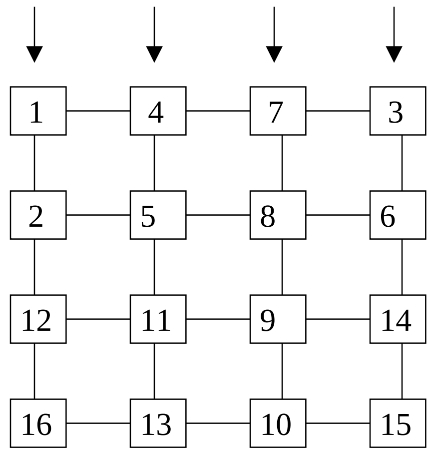
Shearsort - original



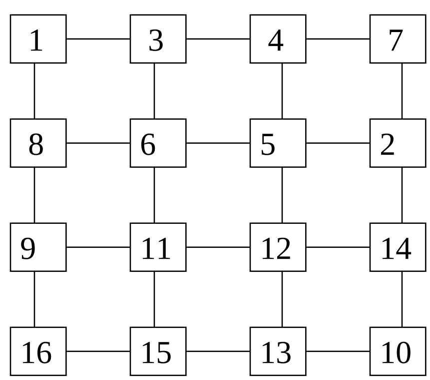
Shearsort: Phase 1 – Row Sort



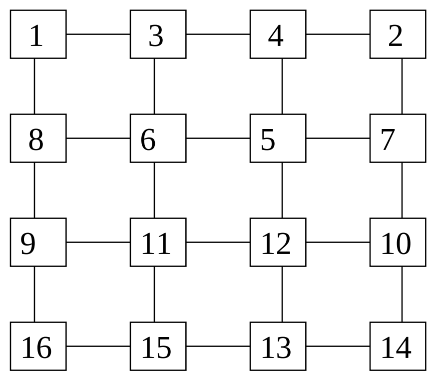
Shearsort: Phase 2 – Column Sort



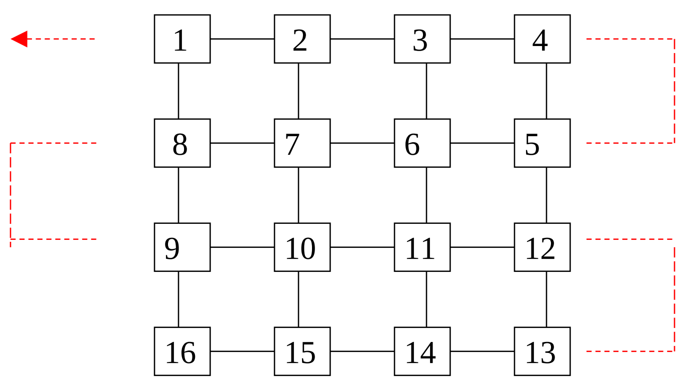
Shearsort: Phase 3 – Row Sort



Shearsort : Phase 4 – Column Sort



Shearsort: Final Phase – Row Sort



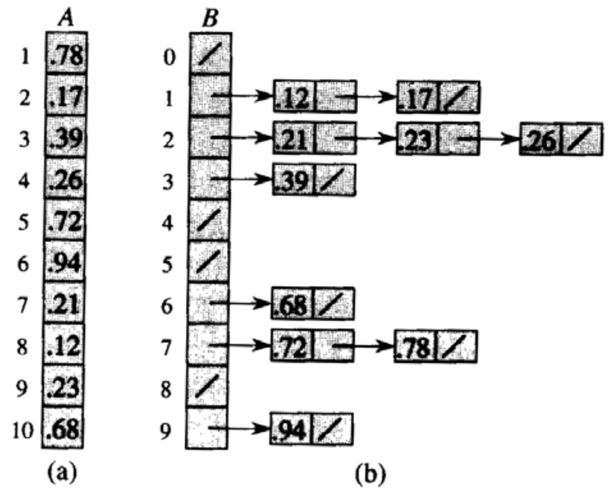
Sorting Applet

- http://www.cs.rit.edu/~atk/Java/Sorting/sorting.html

Other parallel sorts

- Bucket sort
- Pipeline/insertion sort
- Rank sort

Sequential Bucket Sort



Sequential Bucket Sort

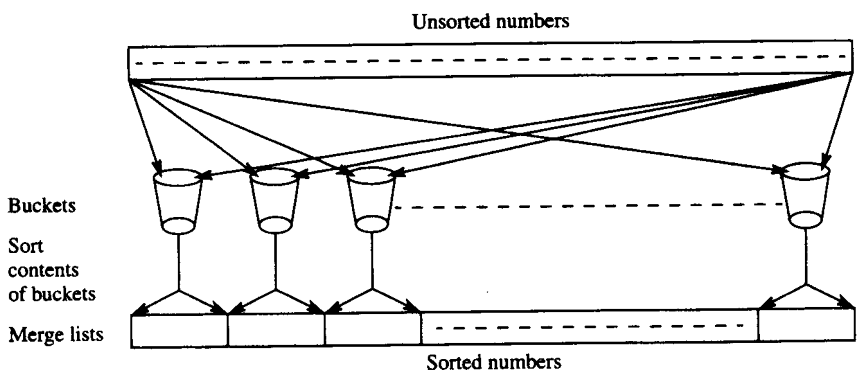


Figure 4.8 Bucket sort.

Sequential Bucket Sort

- Works well only if numbers in list are uniformly distributed across known interval, e.g. *0 – a-1*
- Divide interval into *m* equal (bucket) regions
- Numbers in buckets then sorted using quicksort or mergesort
- Are alternatives, e.g., can recursively divide buckets into smaller buckets => like quicksort without pivot

Sequential Bucket Sort

- Compare each number with start of bucket => *m-1* comparisons per number

OR

- Divide each number by *a*/*m* to get bucket number.
- If *m* power of 2, just need to look at upper bits of number in binary, e.g., assume m is 8 and number in binary is 1100101. This would go into bucket 6 (110) => less expensive then division.

Sequential Bucket Sort

Best case analysis:

- Assume placing a number into bucket requires 1 step and there a n numbers to place. If distribution is uniform n/m numbers/bucket
- Quicksort/mergesort O(n log n), e.g.
  (n/m) log(n/m) here.
- $T_{seq} = n + m \, ((n/m) \log (n/m)) =$
  $n + n \log (n/m) = O(n \log (n/m))$
  if no time needed to compile final sorted list

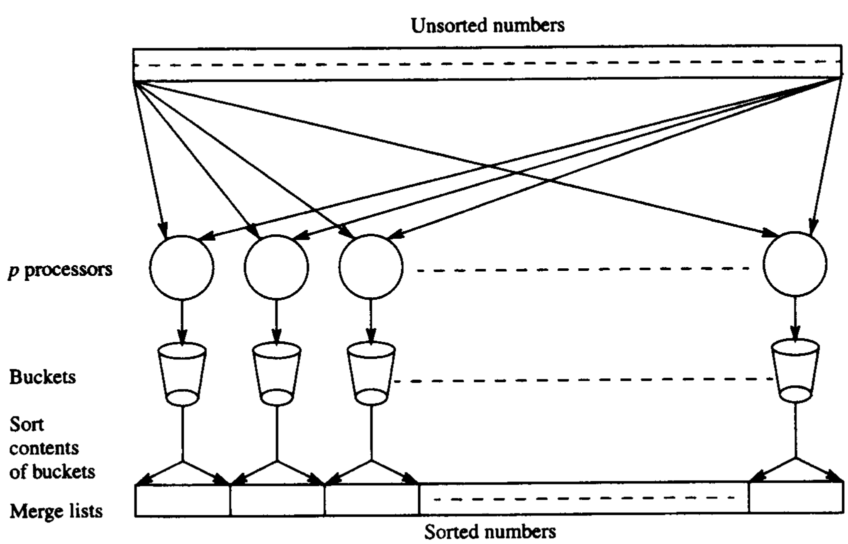


Figure 4.9 One parallel version of bucket sort.

Parallel Bucket Sort (1)

- One bucket per processor
- $T_{par} = n + n/p \log (n/p)$, where n computations needed to place numbers and then numbers are sorted using quicksort.

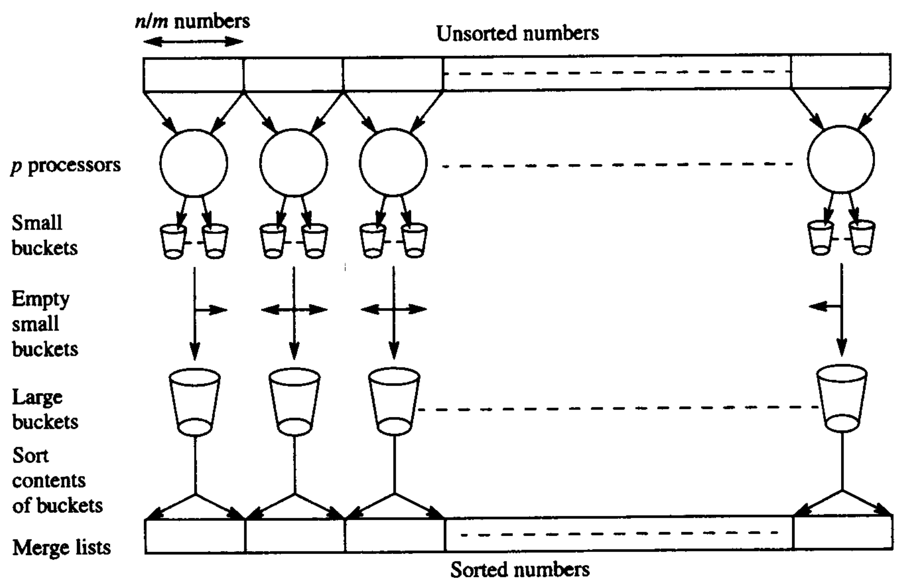


Figure 4.10 Parallel version of bucket sort.

Parallel Bucket Sort (2)

Each processor handles *1/p* of original array and does a bucket sort on it.

Phase 1: Computation and Communication

- n computations needed to partition n numbers into p regions (This seems too big to me!), i.e.,

  $t_{comp1} = n$
- p partitions containing n/p numbers are sent to processes using broadcast or scatter routine, (Here too this seems potentially too big!) i.e.,

  $t_{comm1} = t_{startup} + n\, t_{data}$

Parallel Bucket Sort (2)

Each processor does a bucket sort into p buckets

Phase 2: Computation

- $T_{comp2} = n/p$

Parallel Bucket Sort (2)

If uniform distribution, each small bucket has n/p^2 numbers. Each process must send contents of $p-1$ small buckets to other processes

Phase 3: Communication

- p processors need to make this communication, if cannot overlap

$$t_{comp3} = p\,(p-1)\,(t_{startup} + (n/p^2)\,t_{data})$$

- If communications can overlap, then lower bound is $t_{comm3} = (p-1)\,t_{startup} + (n/p^2)\,t_{data}$

Parallel Bucket Sort (2)

Each processor must sort approximately n/p numbers.

Phase 4: Computation

$$t_{comp4} = (n/p) \log (n/p)$$

Final Cost:

$$T_{\mathrm{par}} = 2t_{startup} + n\,t_{data} + n/p + (p-1)\,t_{startup} + (n/p^2)\,t_{data} + (n/p) \log (n/p)$$

Measures of Performance

- Best sequential (comparison based) sort?
  - O(n log n)
- Time for odd-even sort?
  - O(n)
  - improvement factor of log n faster.
- Other measures?
- What if P<<N?