CS475 Parallel Programming

Sorting

Wim Bohm, Colorado State University

Sorting Problem

- Sorting
  - Input: sequence $S = (a_0, a_1, \ldots, a_{n-1})$
  - Output: $(b_0, b_1, \ldots, b_{n-1})$ = permutation of S s.t. $b_i <= b_{i+1}$
- Sorting Algorithm Categories
  - Internal sorting: S is small enough to fit in memory/network
    - We concentrate on this
  - External sorting: S partly stored on external device (disk)
  - Comparison sorting: Uses compares and exchanges
    - $\Omega(n \log(n))$ work
  - Non comparison sorting: Uses extra information about input data
    - Values lie in a small range (Radix Sort)
    - S is permutation of (1 .. N) (Pigeon hole sort)
    - Sometimes $\Omega(n)$ work

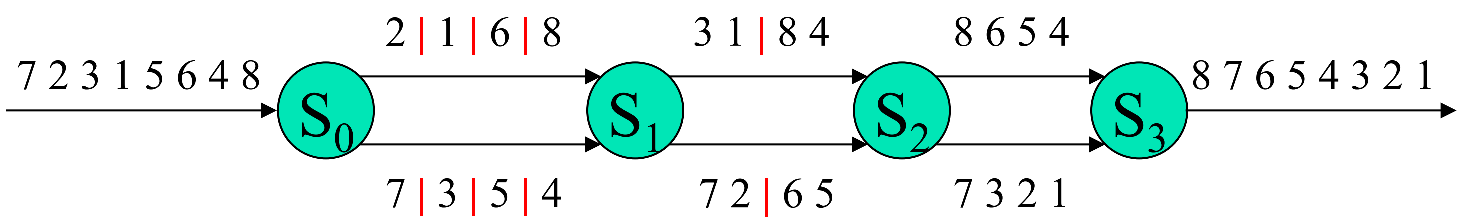
Storage for Input and Output

- Input sequence
  - Distributed in equal blocks over processors unless specified otherwise
- Output sequence
  - Distributed over p processors unless specified otherwise
  - Ordering of blocks S_1 and S_2
    - $S_1 \le S_2$ iff $\forall\, s_1 \in S_1,\ \forall\, s_2 \in S_2$: $s_1 \le s_2$
    - requires enumeration of PEs
      - e.g. on hypercube: PE bit label or Gray code
  - Sorted output
    - $PE_i < PE_j$ (according to enumeration) $\Rightarrow$ $block(PE_i) \le block(PE_j)$

Discussion: Parallel Merge Sort

- A pipeline of sorters $S_0, S_1 \dots. S_n$
- S_0:
  - One input stream, two output streams
  - reads input stream and creates "sorted" subsequences of size 1
  - sends the subsequences to its outputs (alternating between the two)
- S_i: $(i = 1 .. n\text{-}1)$
  - Two input streams, two output streams
  - merges sorted subsequences from two input streams
  - sends double-sized, merged subsequences to its outputs (again alternating)
- S_n:
  - Two input streams, one output stream
  - merges sorted subsequences from two inputs into one result

Parallel Merge Sort (cont.)

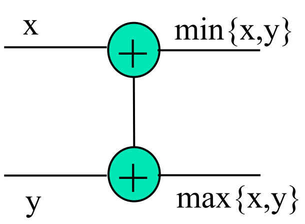


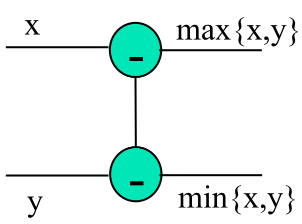
Questions:

1. Given $n = 2^m$ input numbers, how many sorters are needed?
2. If a sorter can read one number in one time step, write one number in one time step, and store and compare in zero time steps, how many time steps does it take to sort n numbers?
3. Is this algorithm cost optimal?

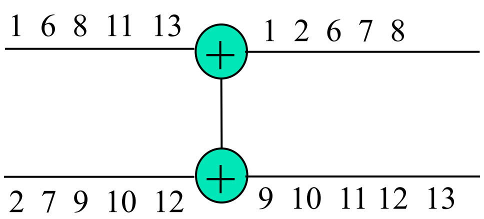
Compare Exchange, Compare Split

- Compare exchange: $n = p$
  - Ascending (+)
  - descending (-)



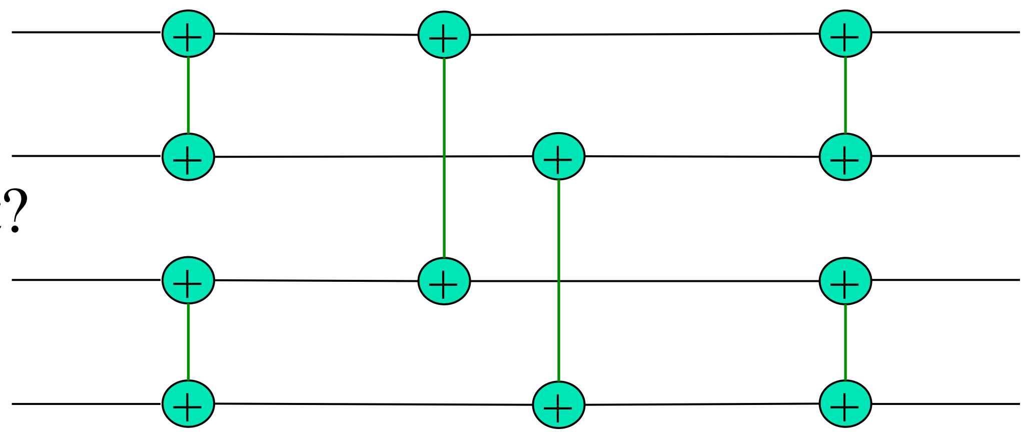


- Compare split: $n > p$
  - P_i and P_j have blocks of data
  - Merge the two blocks
  - P_i gets lower half, P_j gets upper half



Sorting networks

- n numbers, n lines, M stages
- Each stage:
  - <= n/2 compare-exchanges
  - Each compare exchange computes in O(1) time
- time complexity: M
- Cost: M*n
- does this network sort?



Bitonic Sequence

- A sequence $A = a_0, a_1, \ldots, a_{n-1}$ is bitonic iff

  1. There is an index i, $0 < i < n$, s.t.

     $a_0 .. a_i$ is increasing

     and

     $a_i .. a_{n-1}$ is decreasing

  or 2. There is a cyclic shift of A for which 1 holds.

Why called BItonic?

Bitonic Split

- A bitonic split divides a bitonic sequence in two:

$$\text{BitSplit(BS)} = \begin{cases} S1 = (\min(bs_{\mathbf{0}}, bs_{\mathbf{n/2}}),\ \min(bs_{\mathbf{1}}, bs_{\mathbf{n/2+1}}), \ldots, \min(bs_{\mathbf{n/2-1}}, bs_{\mathbf{n-1}})) \\ S2 = (\max(bs_{\mathbf{0}}, bs_{\mathbf{n/2}}),\ \max(bs_{\mathbf{1}}, bs_{\mathbf{n/2+1}}), \ldots, \max(bs_{\mathbf{n/2-1}}, bs_{\mathbf{n-1}})) \end{cases}$$

- Theorem :

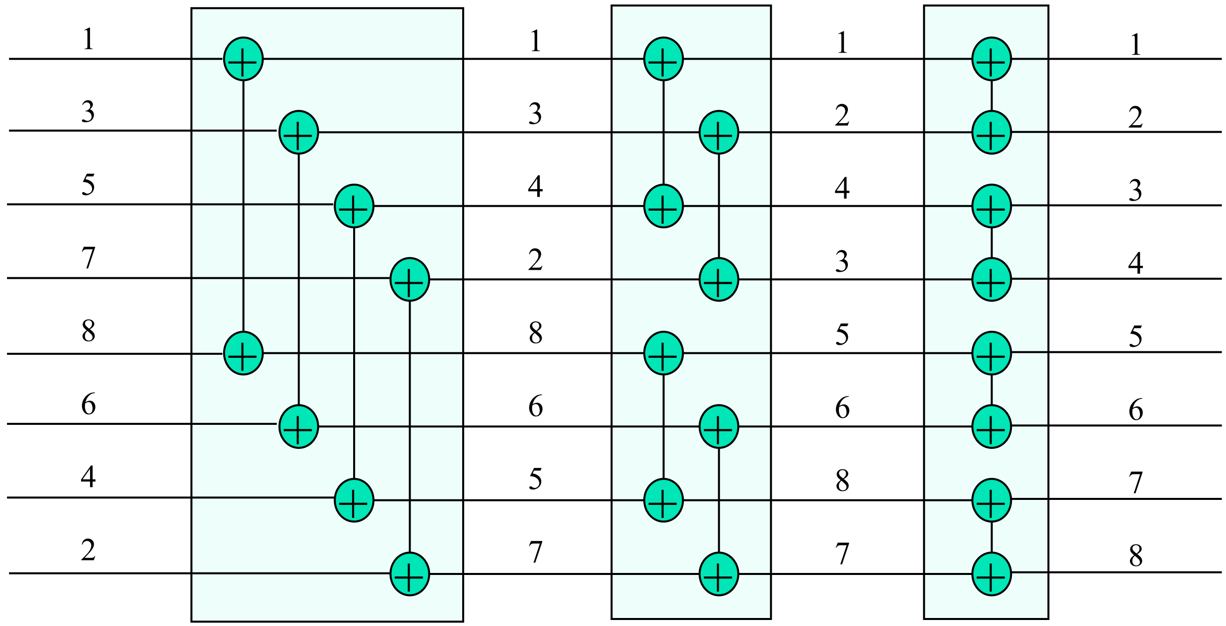
  S1 and S2 are both bitonic and S1 < S2

  Proof:

  By consideration of all cases

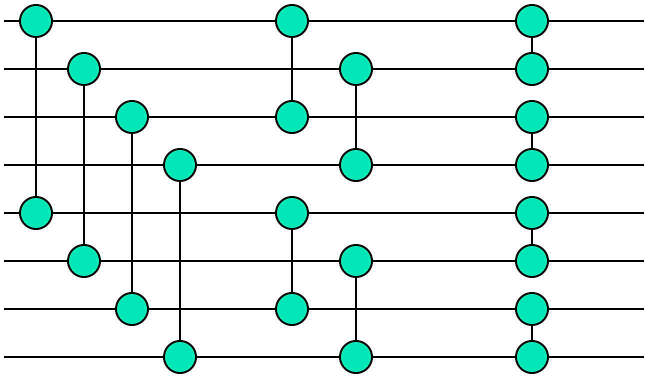
Bitonic Merge

- Given: a Bitonic Sequence BS of size $n = 2^m$
- Sort BS using m (parallel) Bitonic Split stages



Bitonic merge=log(n) bitonic split stages

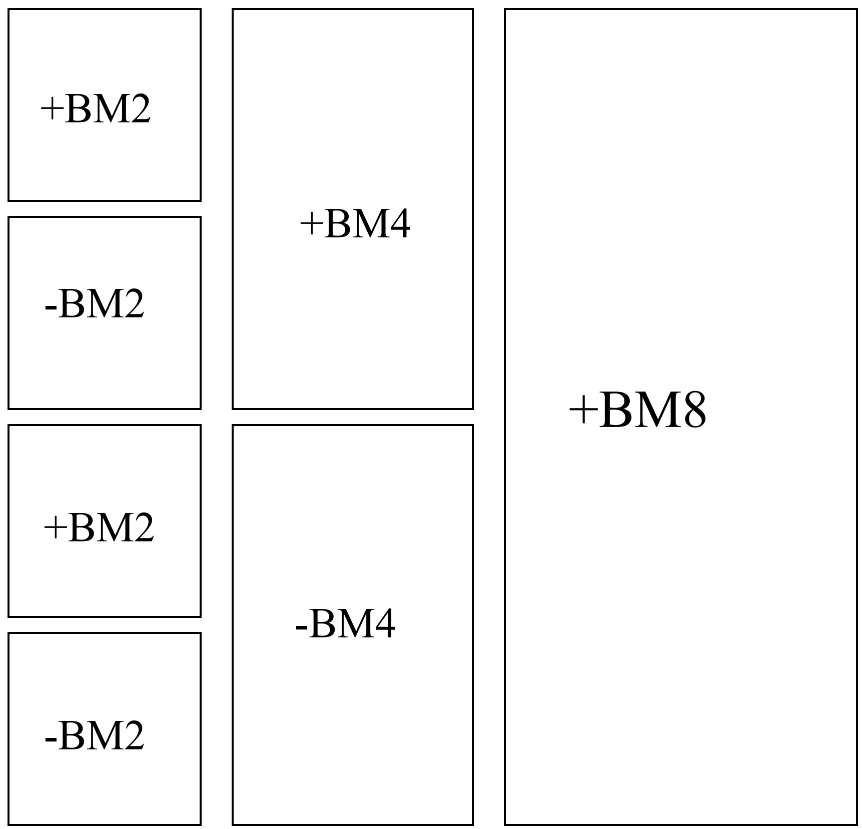
- Can sort a bitonic sequence in log(n) steps
  - Increasing order: +BM(n)
    - use + compare exchangers
  - Decreasing order: -BM(n)
    - use - compare exchangers



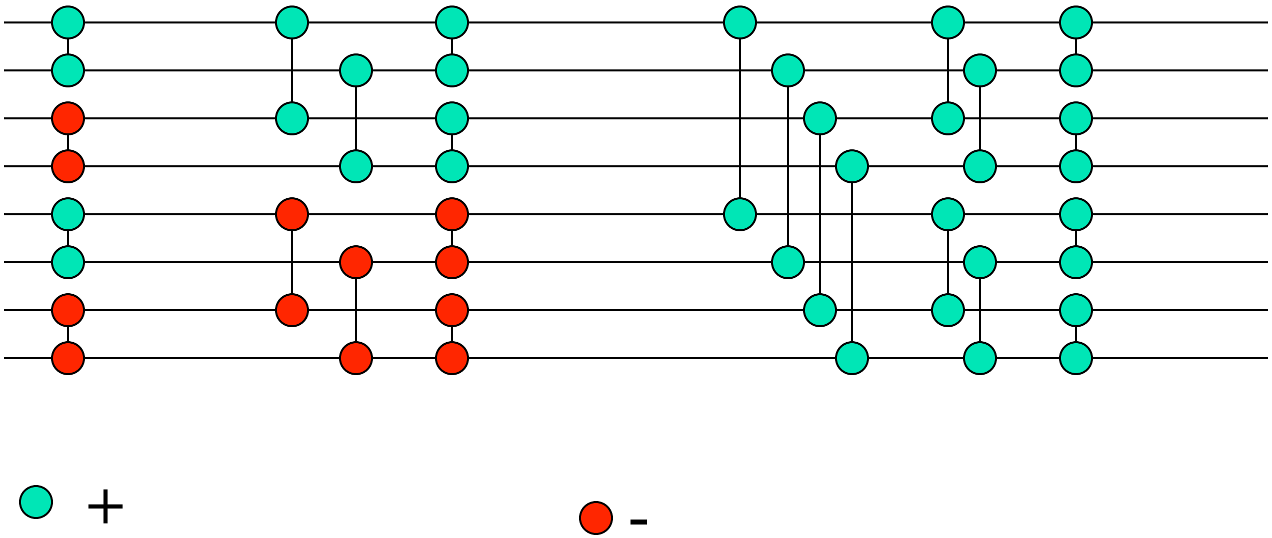
Bitonic Sort

- Each 2 element subsequence is bitonic
- Merging 2 element subsequences, up and down, creates bitonic subsequences of size 4
  - Merging 2 elements up: +BM2
  - Merging 2 elements down: -BM2
- Merging these 4 sized subsequences up (+BM4) and down (-BM4) creates bitonic subsequences of size 8
- and so on......

Bitonic sort = log(n) bitonic merge stages



Bitonic Sort network



Bitonic sort: time and work

- Time: $O(\log^2(n))$

  Number of stages:

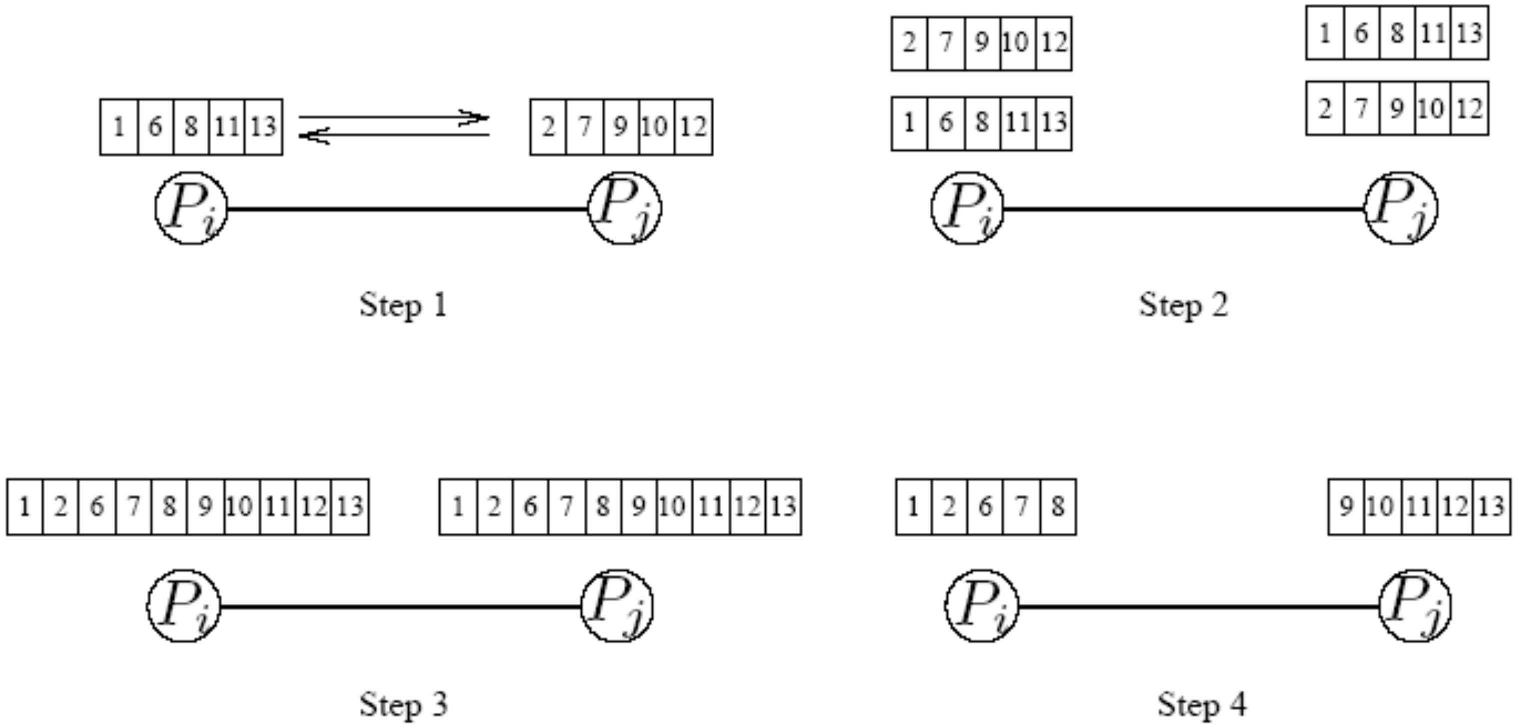
  $B2+B4+B8+\ldots+B2^m = 1 + 2 + 3 + .. + m$

  where $m = \log(n)$

- Work: $O(n \log^2(n))$

  $O(n)$ per stage

Sorting: Parallel Compare Split Operation

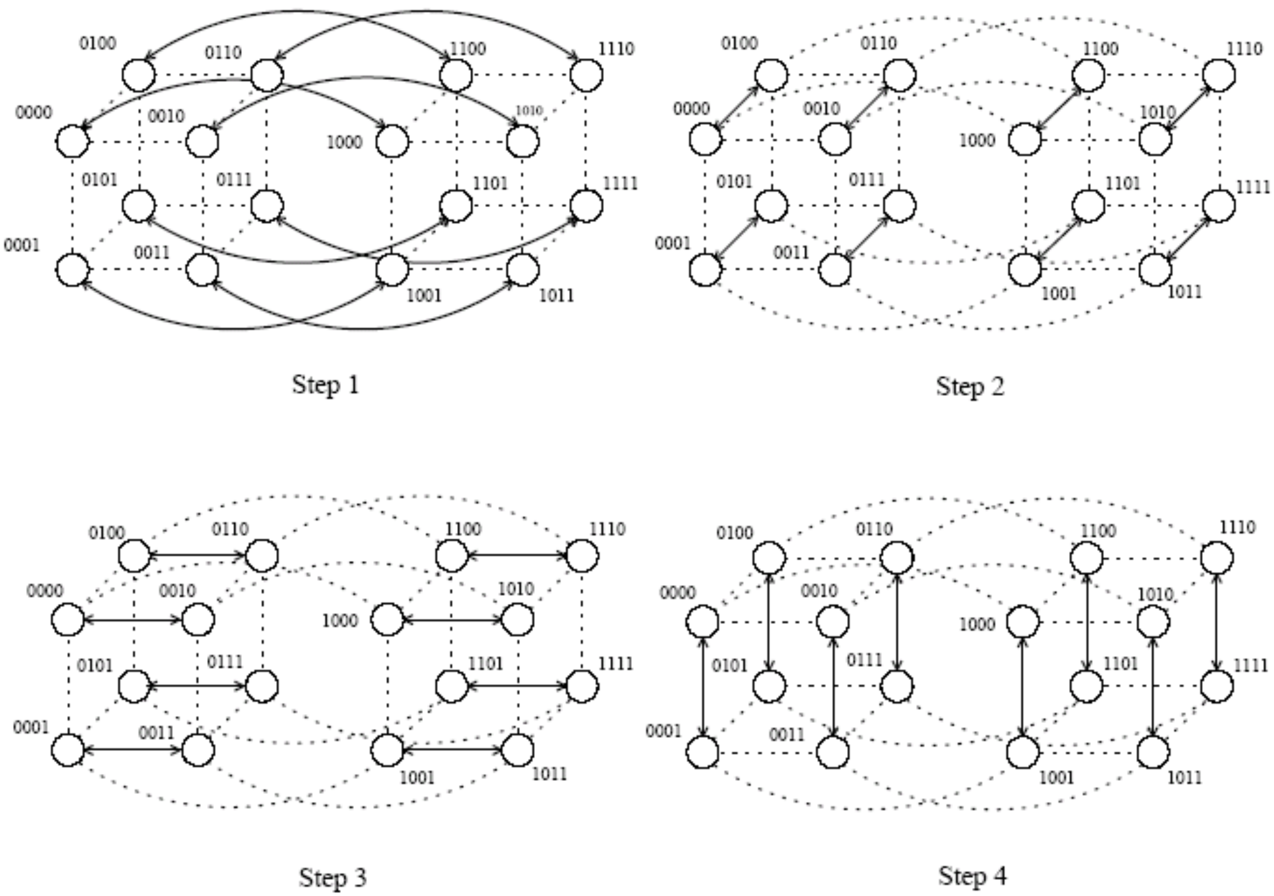


A compare-split operation. Each process sends its block of size n/p to the other process. Each process merges the received block with its own block and retains only the appropriate half of the merged block. In this example, process P_i retains the smaller elements and process P_i retains the larger elements.

Mapping Bitonic Sort to Hypercubes

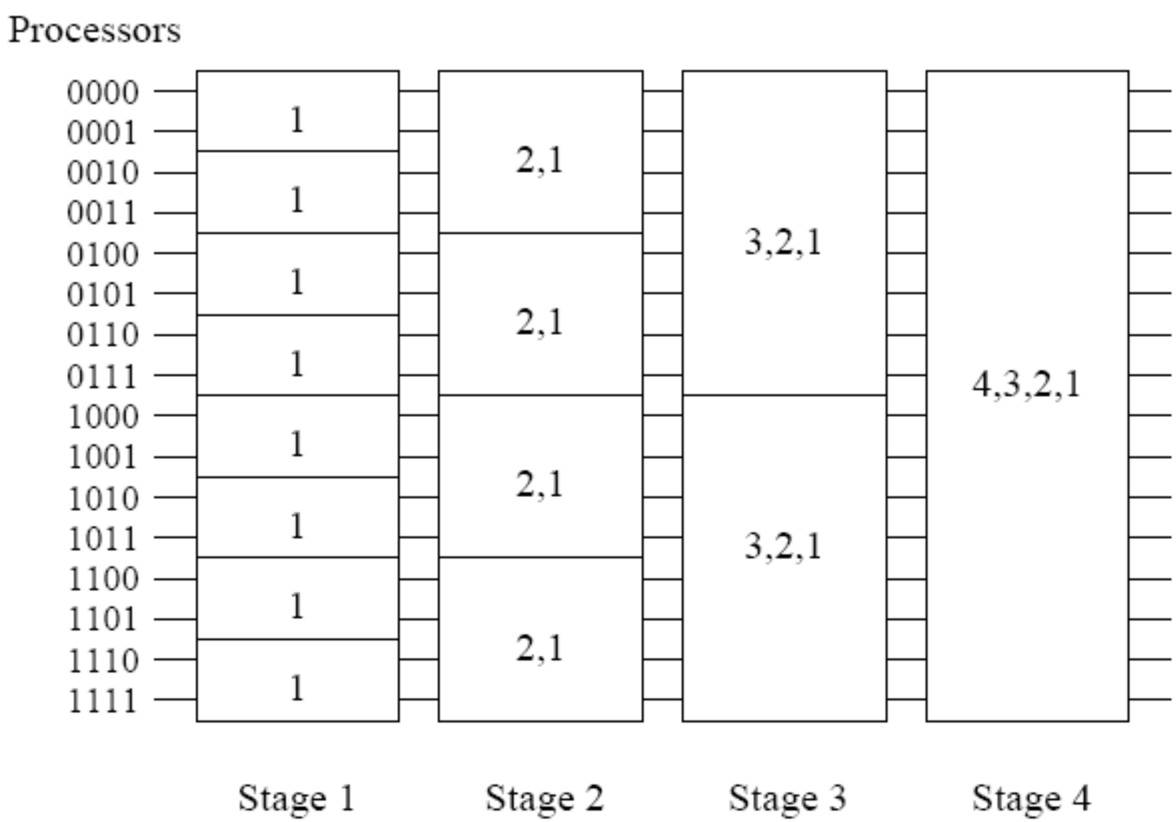
- Consider the case of one item per processor. The question becomes one of how the wires in the bitonic network should be mapped to the hypercube interconnect.
- Note from our earlier examples that the compare-exchange operation is performed between two wires only if their labels differ in exactly one bit!
- This implies a direct mapping of wires to processors. All communication is nearest neighbor!

Mapping Bitonic Sort to Hypercubes



Communication during the last stage of bitonic sort. Each wire is mapped to a hypercube process; each connection represents a compare-exchange between processes.

Mapping Bitonic Sort to Hypercubes



Communication characteristics of bitonic sort on a hypercube. During each stage of the algorithm, processes communicate along the dimensions shown.

Mapping Bitonic Sort to Hypercubes

```
procedure BITONIC_SORT(label, d)
begin
    for i := 0 to d - 1 do
        for j := i downto 0 do
            if (i + 1)^st bit of label ≠ j^th bit of label then
                comp_exchange_max(j);
            else
                comp_exchange_min(j);
end BITONIC_SORT
```

Parallel formulation of bitonic sort on a hypercube with $n = 2^d$ processes.

Mapping Bitonic Sort to Hypercubes

- During each step of the algorithm, every process performs a compare-exchange operation (single nearest neighbor communication of one word).
- Since each step takes $\Theta(1)$ time, the parallel time is

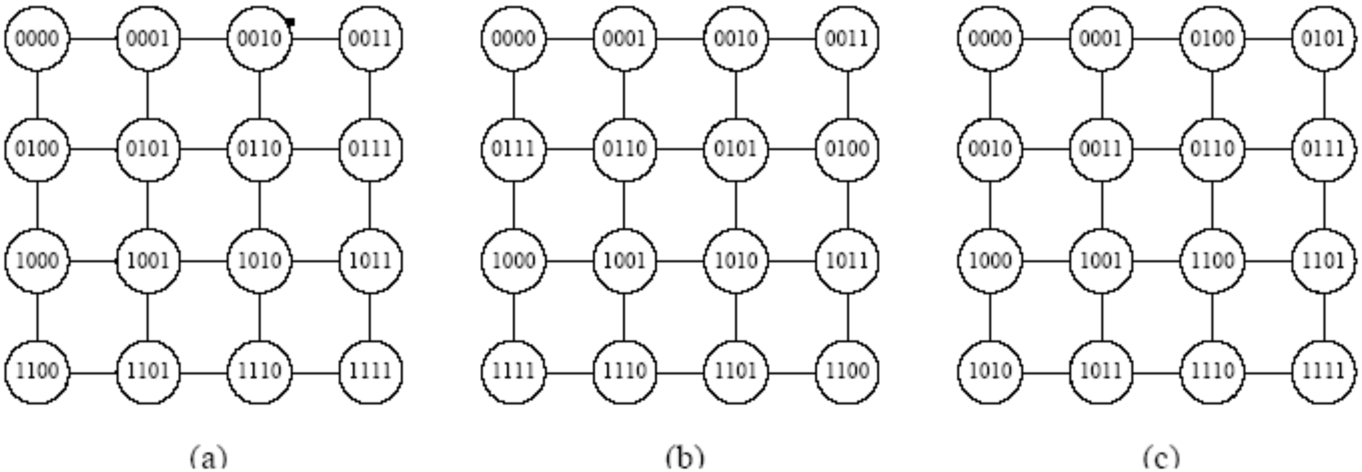
$$T_p = \Theta(\log^2 n) \qquad (2)$$

- This algorithm is cost optimal w.r.t. its serial counterpart, but not w.r.t. the best sorting algorithm.

Mapping Bitonic Sort to Meshes

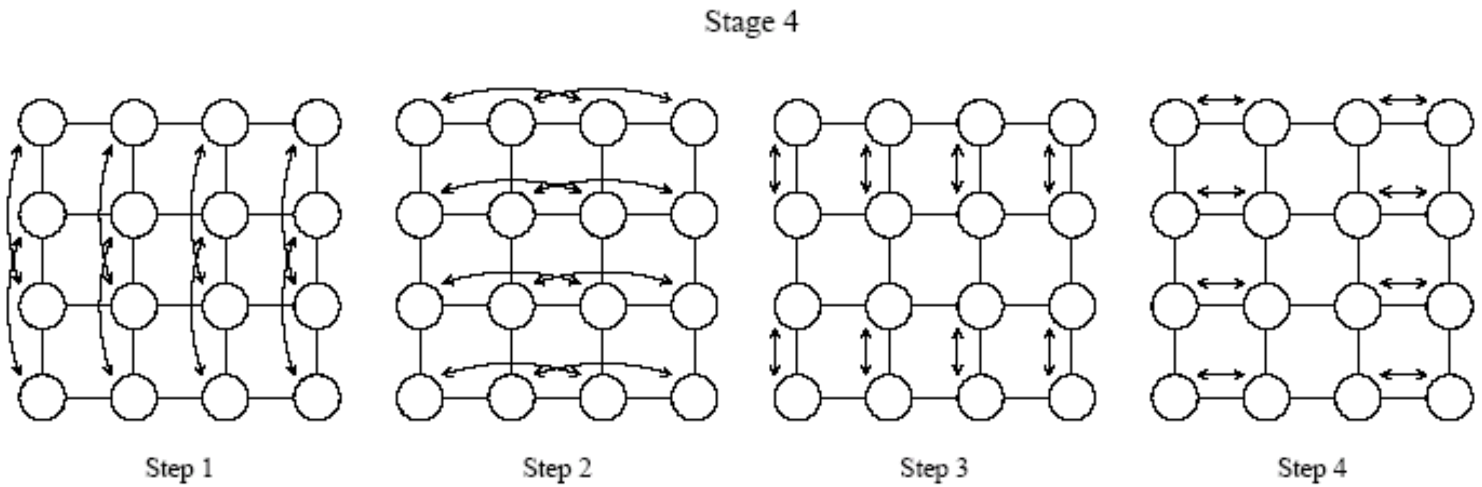
- The connectivity of a mesh is lower than that of a hypercube, so we must expect some overhead in this mapping.
- Consider the row-major shuffled mapping of wires to processors.

Mapping Bitonic Sort to Meshes



Different ways of mapping the input wires of the bitonic sorting network to a mesh of processes: (a) row-major mapping, (b) row-major snakelike mapping, and (c) row-major shuffled mapping.

Mapping Bitonic Sort to Meshes



The last stage of the bitonic sort algorithm for $n = 16$ on a mesh, using the row-major shuffled mapping. During each step, process pairs compare-exchange their elements. Arrows indicate the pairs of processes that perform compare-exchange operations.

Mapping Bitonic Sort to Meshes

- In the row-major shuffled mapping, wires that differ at the i^{th} least-significant bit are mapped onto mesh processes that are $2^{\lfloor (i-1)/2 \rfloor}$ communication links away.
- The total amount of communication performed by each process is $\sum_{i=1}^{\log n} \sum_{j=1}^{i} 2^{\lfloor (j-1)/2 \rfloor} \approx 7\sqrt{n}$, or $\Theta(\sqrt{n})$. The total computation p is $\Theta(\log^2 n)$.

$$T_P = \overbrace{\Theta(\log^2 n)}^{\text{comparisons}} + \overbrace{\Theta(\sqrt{n})}^{\text{communication}}.$$

- The parallel runtime is:

Block of Elements Per Processor

- Each process is assigned a block of n/p elements.
- The first step is a local sort of the local block.
- Each subsequent compare-exchange operation is replaced by a compare-split operation.
- We can effectively view the bitonic network as having $(1 + \log p)(\log p)/2$ steps.

Block of Elements Per Processor: Hypercube

- Initially the processes sort their n/p elements (using merge sort) in time $\Theta((n/p)\log(n/p))$ and then perform $\Theta(\log^2 p)$ compare-split steps.
- The parallel run time of this formulation is

$$T_P = \overbrace{\Theta\left(\frac{n}{p}\log\frac{n}{p}\right)}^{\text{local sort}} + \overbrace{\Theta\left(\frac{n}{p}\log^2 p\right)}^{\text{comparisons}} + \overbrace{\Theta\left(\frac{n}{p}\log^2 p\right)}^{\text{communication}}.$$

- Comparing to an optimal sort, the algorithm can efficiently use up to $p = \Theta(2^{\sqrt{\log n}})$ processes.
- The isoefficiency function due to both communication and extra work is $\Theta(p^{\log p}\log^2 p)$.

Block of Elements Per Processor: Mesh

- n by:

$$T_P = \overbrace{\Theta\left(\frac{n}{p}\log\frac{n}{p}\right)}^{\text{local sort}} + \overbrace{\Theta\left(\frac{n}{p}\log^2 p\right)}^{\text{comparisons}} + \overbrace{\Theta\left(\frac{n}{\sqrt{p}}\right)}^{\text{communication}}$$

- This formulation can efficiently use up to $p = \Theta(\log^2 n)$ processes.
- The isoefficiency function is $\Theta(2^{\sqrt{p}}\sqrt{p})$.

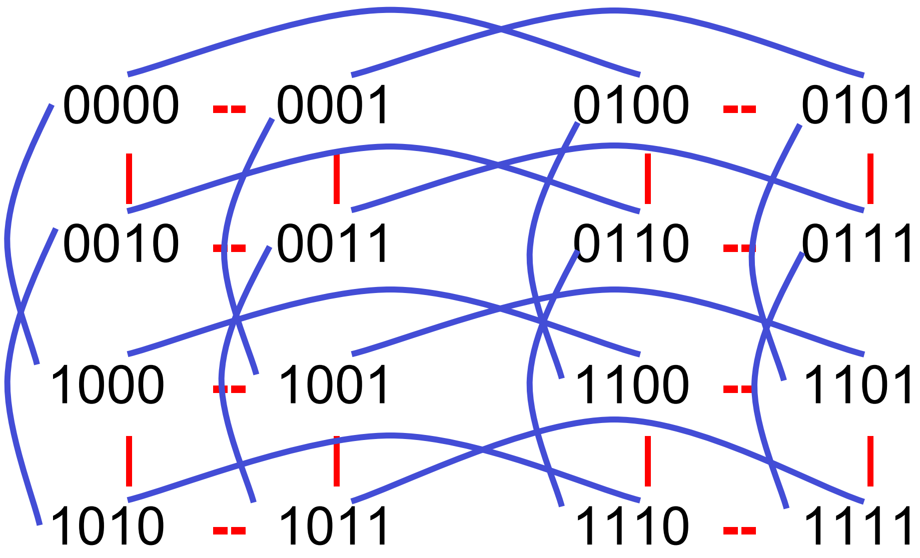
Performance of Parallel Bitonic Sort

Architecture	Maximum Number of Processes for $E = \Theta(1)$	Corresponding Parallel Run Time	Isoefficiency Function
Hypercube	$\Theta(2^{\sqrt{\log n}})$	$\Theta(n/(2^{\sqrt{\log n}}) \log n)$	$\Theta(p^{\log p} \log^2 p)$
Mesh	$\Theta(\log^2 n)$	$\Theta(n/ \log n)$	$\Theta(2^{\sqrt{p}} \sqrt{p})$
Ring	$\Theta(\log n)$	$\Theta(n)$	$\Theta(2^p p)$

The performance of parallel formulations of bitonic sort for *n* elements on *p* processes.

Bitonic Sort on Mesh

- No ideal mapping; best: nearest = most used



Distance 1: used 7 times, Distance 2: used 3 times

Bitonic Sort n > p

- n/p elements per PE
- Do local sorts at the beginning
- Use compare-split instead of compare-exchange
- Perfect load balance

Parallel Bubble Sort

Odd-Even sort:

- sorts n elements in n/2 phases
- Each phase has two stages
  - first stage compares even element with next element
  - second stage compares odd element with next
- O(n) time, O(n^2) work

Count/Radix/Bucket family

- Enumeration Sort
  - Determine rank of every element
  - Sort A[0..n-1], using counters C[0..n-1]

```
forall i in 0..n-1 C[i]=0
forall i in 0..n-1, forall j in 0..n-1
    if A[i]<A[j] or (A[i]==A[j] and i<j) C[j]++
forall i in 0..n-1 S[C[i]]=A[i]
```

Count sort: large number of small numbers

- n numbers in range 0..r-1
  - n >> r

forall i in 0..r C[i]=0
forall i in 0..n-1 C[A[i]+1]++
PPC = ParallelPrefixSum(C)
forall i in 0..n-1 S[PPC[A[i]]++]=A[i]

- One of the fastest sorts for this case

Partial sums, or Parallel Prefix

N numbers V_1 to V_n stored in A[1] to A[n]

Compute all partial sums $(V_1+..+V_k)$

```
d = 1
do log(n) times
    for all i in 1..n:
        if (i-d)>0 A[i] = A[i]+A[i-d]
    d *= 2
```