Parallel Systems Course: Chapter VIII

Sorting Algorithms

Kumar Chapter 9

Jan Lemeire
ETRO Dept.
Fall 2017

Vrije Universiteit Brussel

Overview

1. Parallel sort – distributed memory
2. Parallel sort – shared memory
3. Sorting Networks
   - A. Odd-even
   - B. Bitonic
4. Parallel sort - GPU

Sorting: Overview

- One of the most commonly used and well-studied kernels.
- Sorting can be *comparison-based* or *non-comparison-based*.
  - Non-comparison: determine rank (index) in list of element
    - E.g. Radix sort: put elements in buckets
- We focus here on comparison-based sorting algorithms.
  - The fundamental operation of comparison-based sorting is *compare-exchange*.
  - The lower bound on any comparison-based sort of n numbers is $\Theta(n \log n)$, the quicksort performance.

Overview

1. Parallel sort – distributed memory
2. Parallel sort – shared memory
3. Sorting Networks
   A. Odd-even
   B. Bitonic
4. Parallel sort - GPU

Mission

Sort array *asap* by exploiting parallel system with distributed memory

- *Idea*: based on quicksort
  - *leads to most-optimal parallel algorithm?*

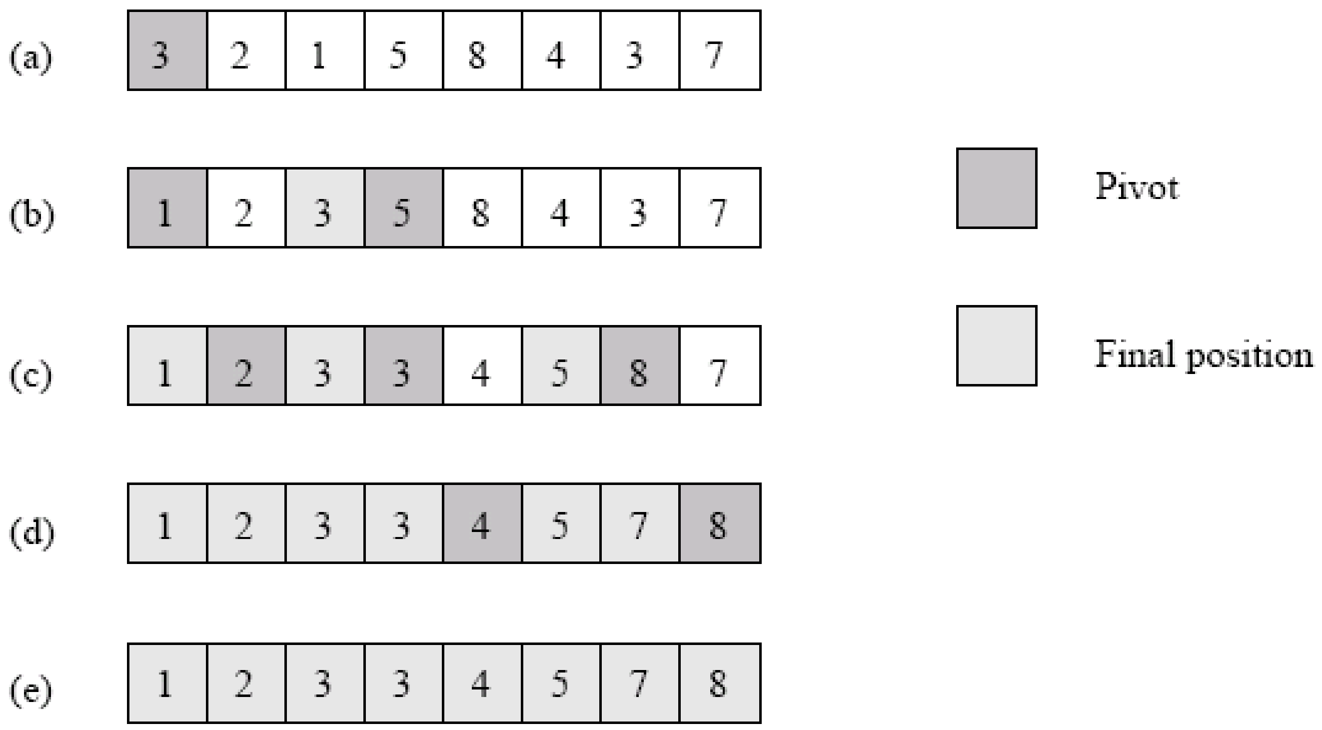
Quicksort

- Quicksort is one of the most common sorting algorithms for sequential computers because of its simplicity, low overhead, and optimal average complexity.
- Quicksort selects one of the entries in the sequence to be the pivot and divides the sequence into two - one with all elements less than the pivot and other greater.
- The process is recursively applied to each of the sublists.

```
procedure QUICKSORT (A, q, r)
begin
    if q < r then
    begin
        x := A[q];
        s := q;
        for i := q + 1 to r do
            if A[i] ≤ x then
            begin
                s := s + 1;
                swap(A[s], A[i]);
            end if
        swap(A[q], A[s]);
        QUICKSORT (A, q, s);
        QUICKSORT (A, s + 1, r);
    end if
end QUICKSORT
```

The sequential quicksort algorithm.

Quicksort



Example of the quicksort algorithm sorting a sequence of size $n = 8$.

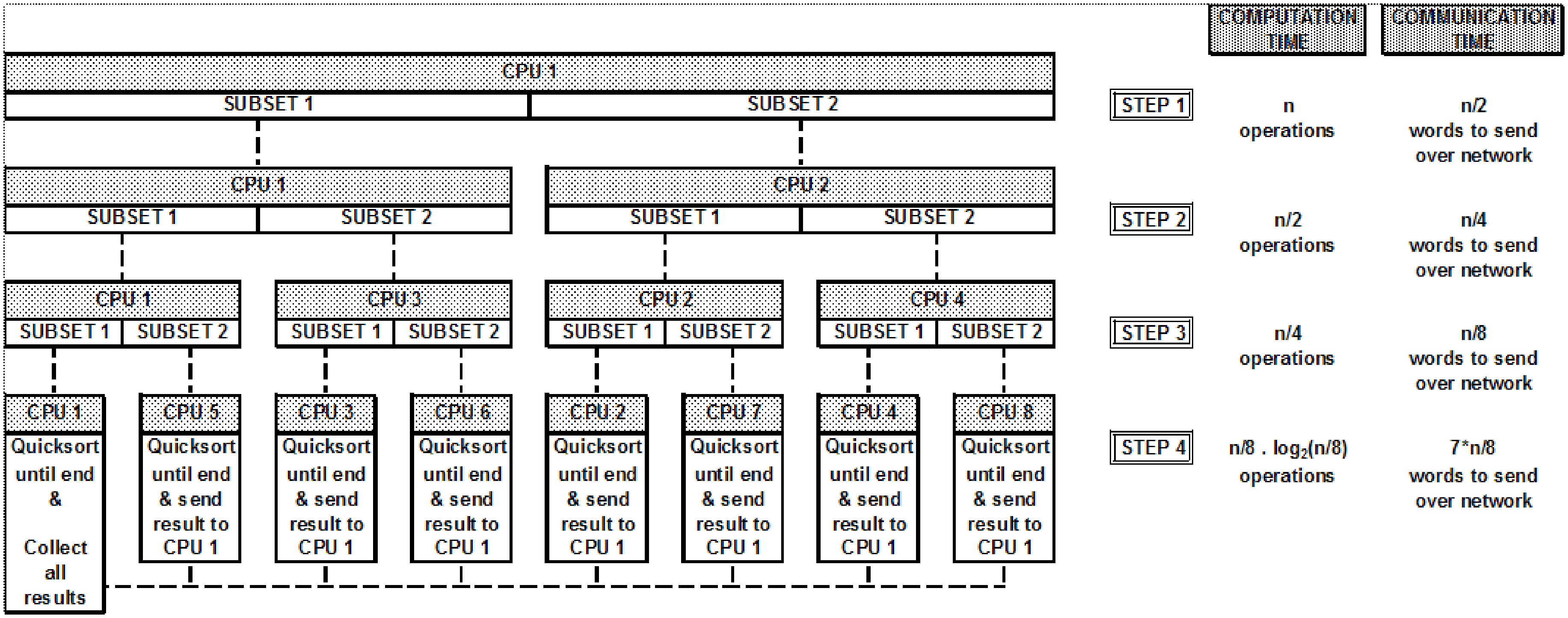
Quicksort

- The performance of quicksort depends critically on the quality of the pivot.
- In the best case, the pivot divides the list in such a way that the larger of the two lists does not have more than αn elements (for some constant α).
- In this case, the complexity of quicksort is $\boldsymbol{O(n \log n)}$.

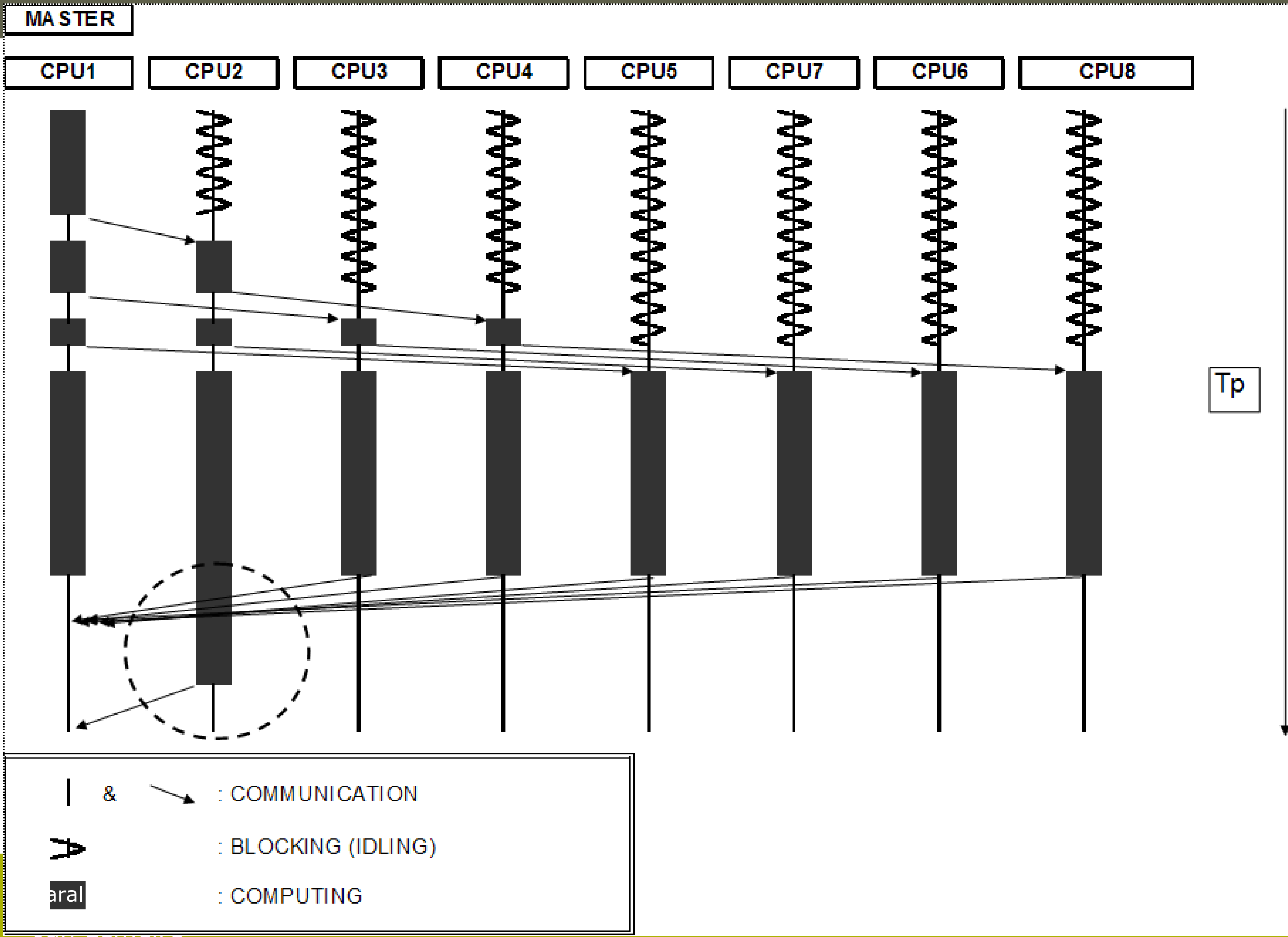
v1. Parallel Quicksort

- Lets start with recursive decomposition - the list is partitioned serially and each of the subproblems is handled by a different processor.
- The time for this algorithm is lower-bounded by $\Omega(n)$!
  - Since the partitioning is done on single processor
- Can we parallelize the partitioning step - in particular, if we can use n processors to partition a list of length n around a pivot in $O(1)$ time, we have a winner.
  - Then we obtain a runtime of $O(\log n)$!!
- This is difficult to do on real machines, though.

Parallel Quicksort



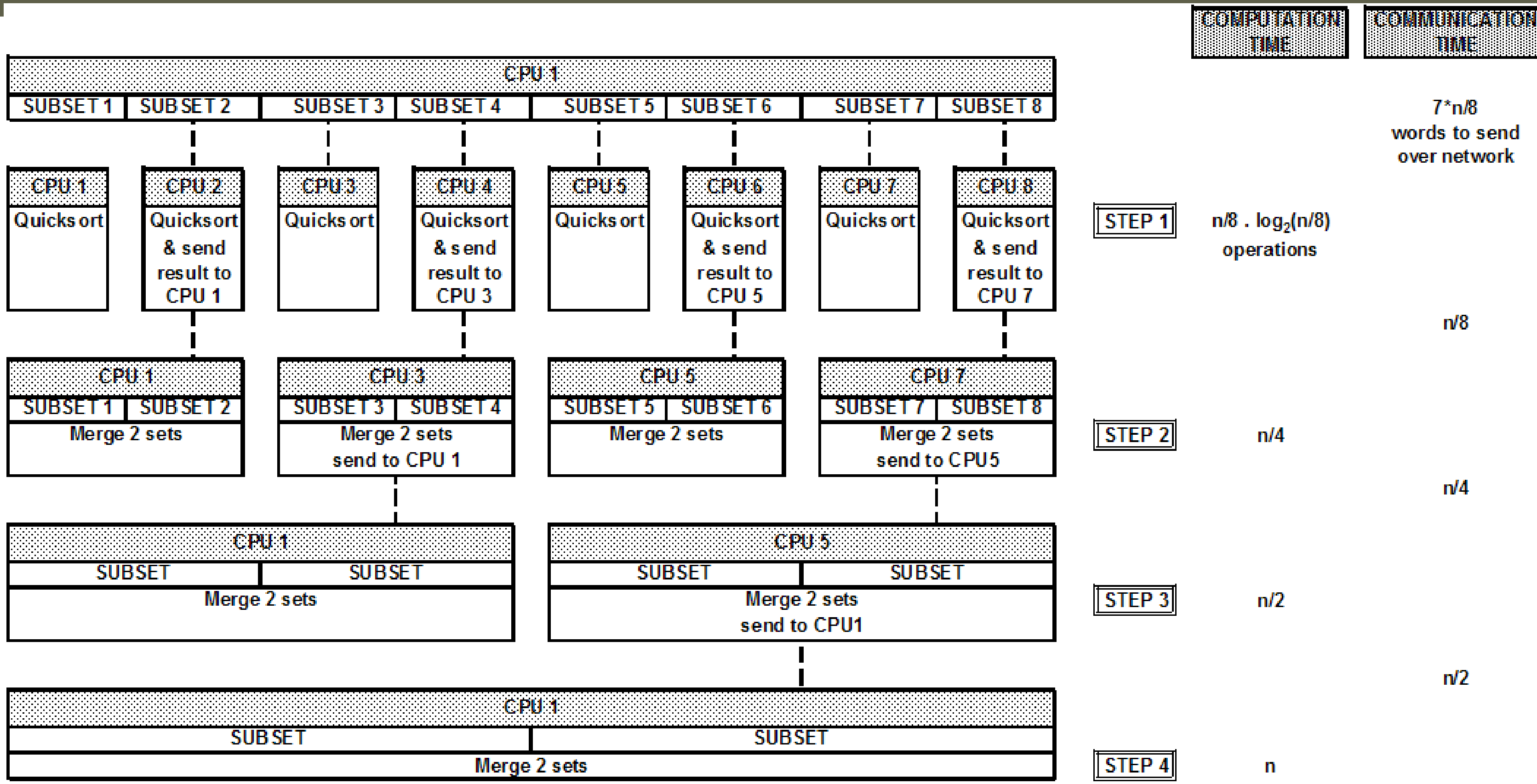
Execution Profile



Can we resolve the load imbalances?

- Make sure that each processor has the same number of elements locally.
- Merge results
- Merge sort!
  - Actually better than quicksort
  - Disadvantage: not in place (need copy of matrix)
    - Use quicksort for local sort

v2. based on merge sort

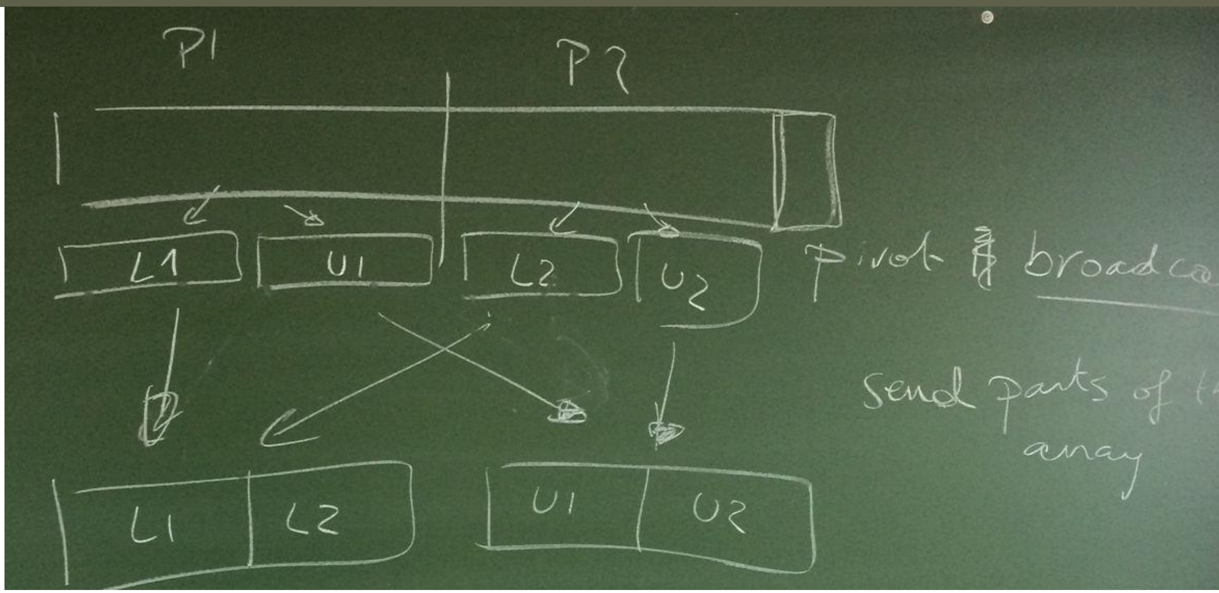


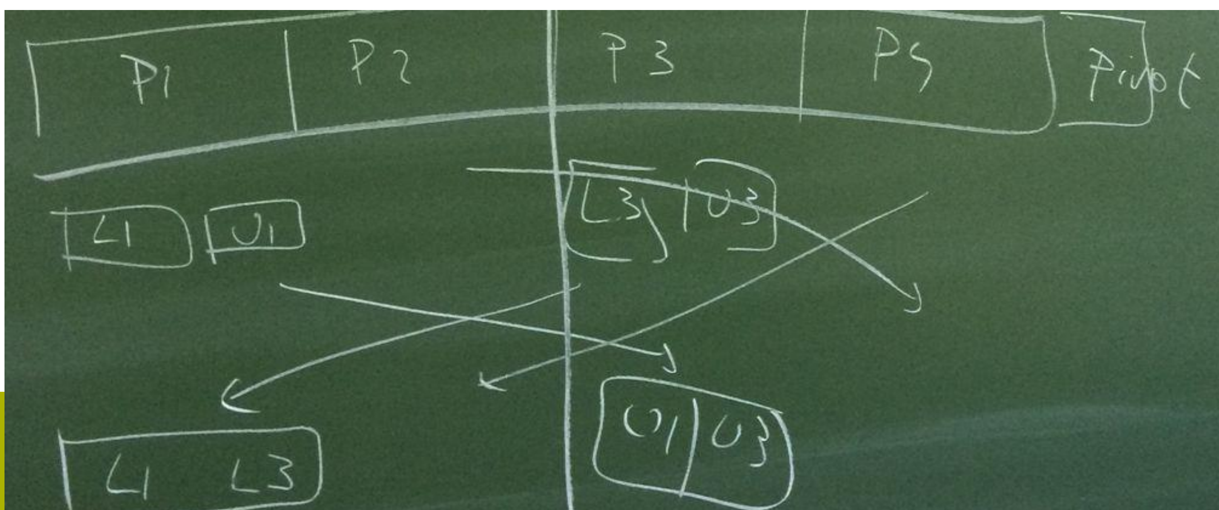
Similar communication overhead, but without load imbalances!

v3. Can we overcome the limited parallelism in the beginning?

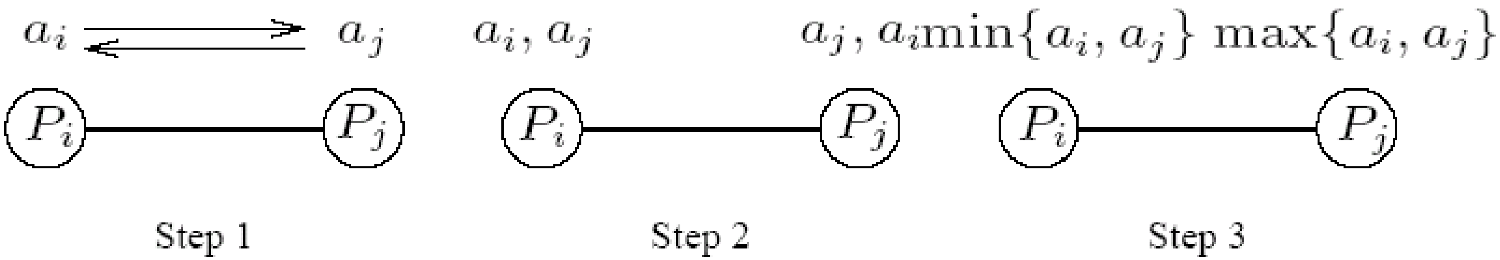
- A simple message passing formulation is based on the recursive halving of the machine.
- Assume that each processor in the lower half of a p processor ensemble is paired with a corresponding processor in the upper half.
- A designated processor selects and broadcasts the pivot.
- Each processor splits its local list into two lists, one less (L_i), and other greater (U_i) than the pivot.
- A processor in the low half of the machine sends its list U_i to the paired processor in the other half. The paired processor sends its list L_i.

Halving process in parallel





Sorting: Parallel Compare Exchange Operation

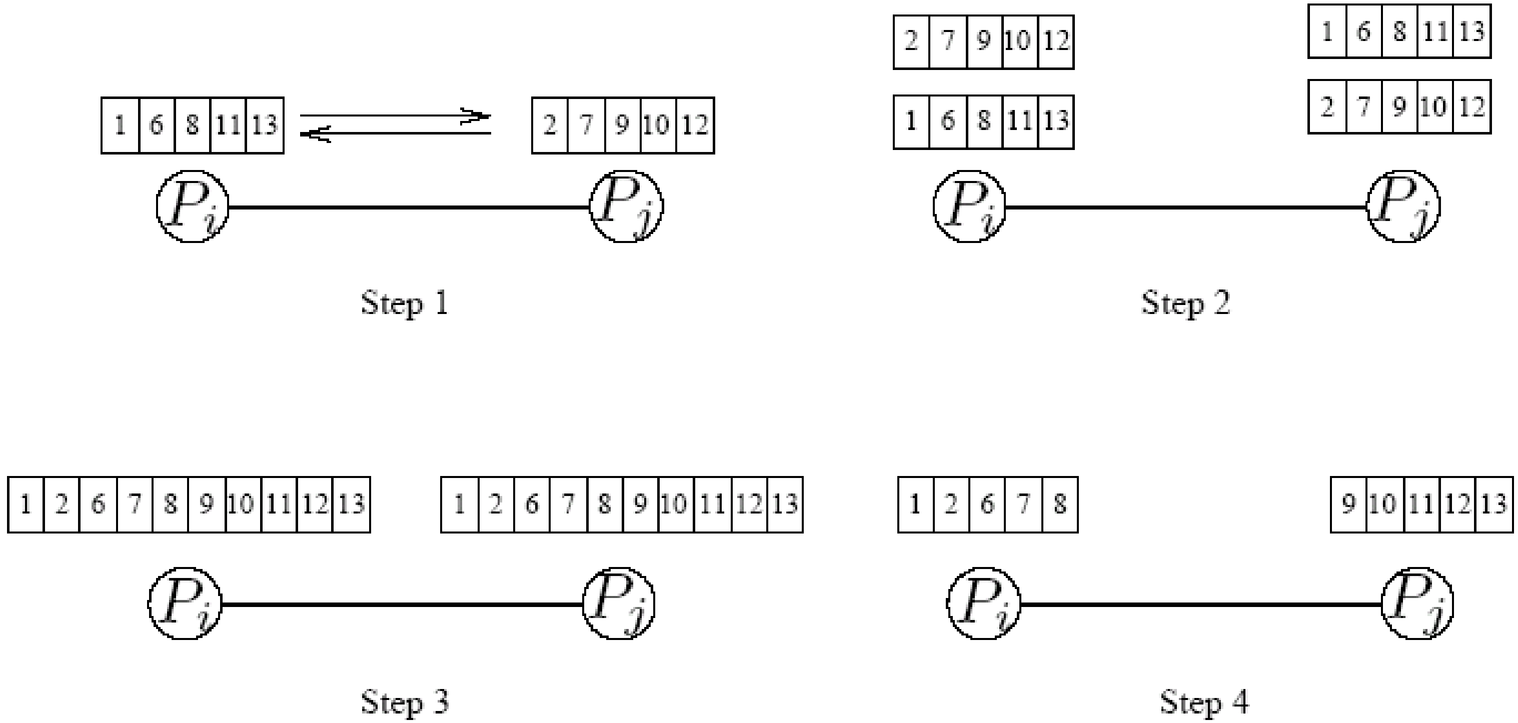


A parallel **compare-exchange** operation. Processes P_i and P_j send their elements to each other. Process P_i keeps $\min\{a_i,a_j\}$, and P_j keeps $\max\{a_i, a_j\}$.

Extending to n/p elements

What is the parallel counterpart to a sequential comparator?

- If each processor has one element, the compare exchange operation can be done in $t_s + t_w$ time (startup latency and per-word time).
- If we have more than one element per processor, we call this operation a **compare split**. Assume each of two processors have n/p elements.
  - After the compare-split operation, the smaller n/p elements are at processor P_i and the larger n/p elements at P_j, where $i < j$.
  - The time for a compare-split operation is $(t_s + t_w n/p)$, assuming that the two partial lists were initially sorted.



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.

There are alternatives! With more communication, however…

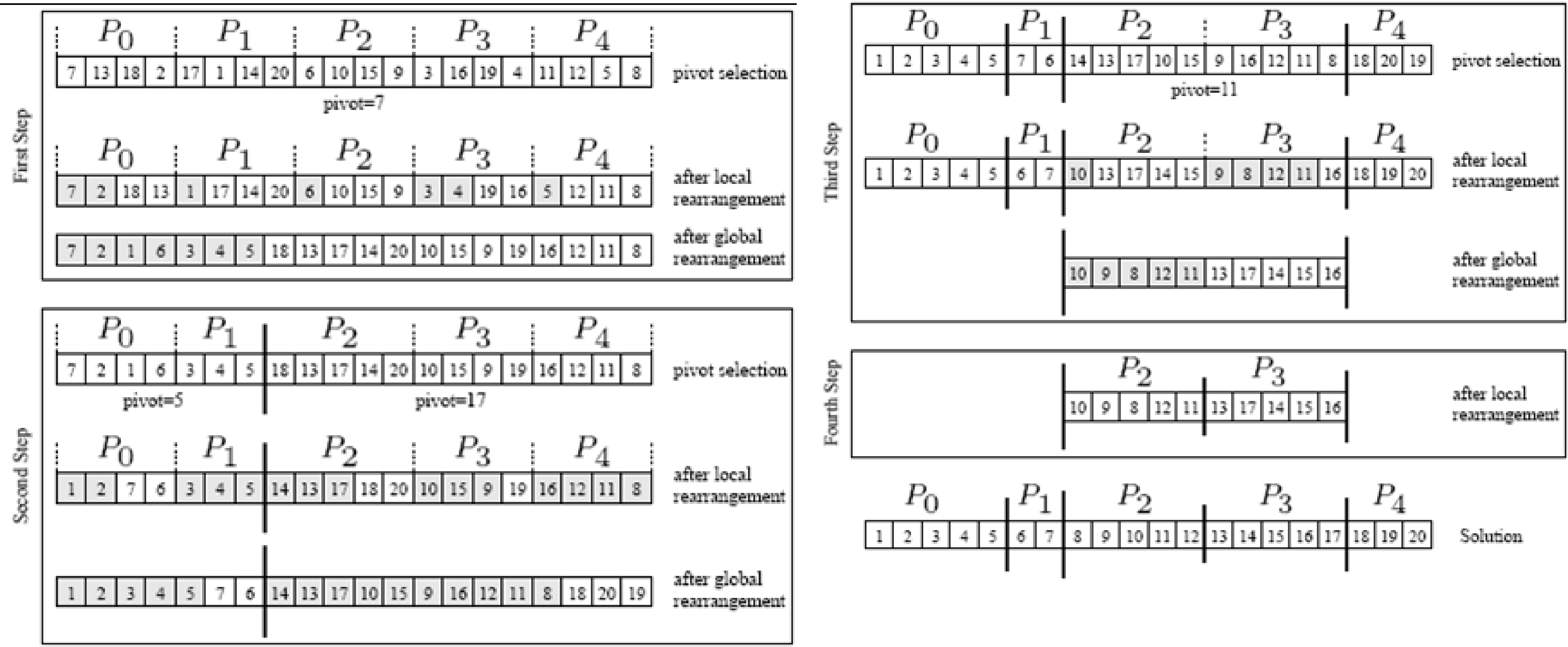
- After this step:
  - all elements < pivot in the low half of the machine
  - all elements > pivot in the high half.
- The above process is recursed until each processor has its own local list, which is sorted locally.
- The time for a single reorganization is $\Theta(\log p)$ for broadcasting the pivot element, $\Theta(n/p)$ for splitting the locally assigned portion of the array, $\Theta(n/p)$ for exchange and local reorganization.
- Note that this time is identical to that of the corresponding shared address space formulation.
- However, it is important to remember that the reorganization of elements is a bandwidth sensitive operation.

Overview

1. Parallel sort – distributed memory
2. **Parallel sort – shared memory**
3. Sorting Networks
   A. Odd-even
   B. Bitonic
4. Parallel sort - GPU

Parallelizing Quicksort: Shared Address Space Formulation

- A list of size n equally divided across p processors.
- A pivot is selected by one of the processors and made known to all processors.
- Each processor partitions its list into two, say L_i and U_i, based on the selected pivot.
- All of the L_i lists are merged and all of the U_i lists are merged separately.
- The set of processors is partitioned into two (in proportion of the size of lists L and U). The process is recursively applied to each of the lists.

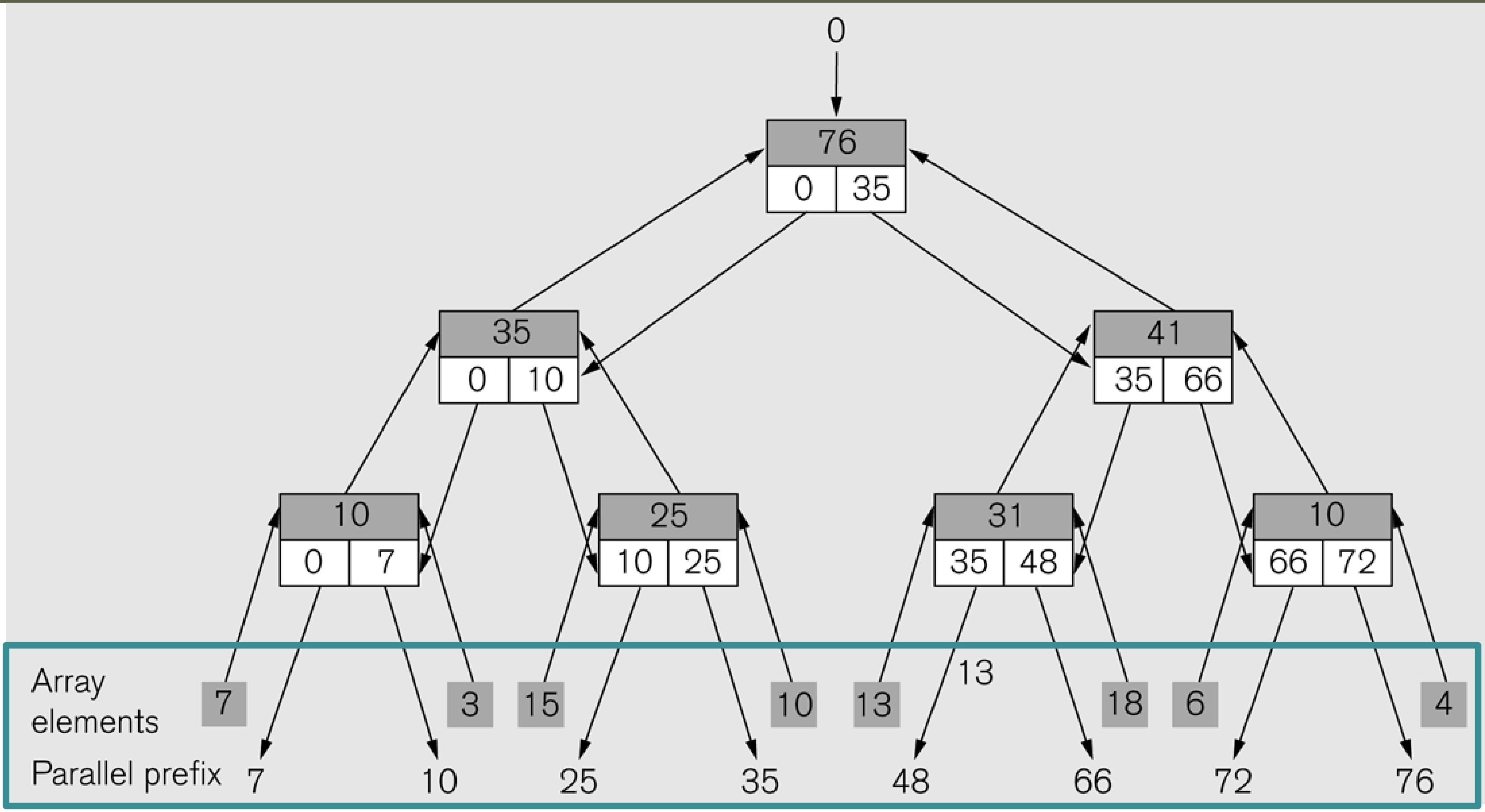

First Step
pivot selection
pivot=7
after local rearrangement
after global rearrangement
Second Step
pivot=5
pivot=17
Third Step
pivot=11
Fourth Step
Solution

Parallelizing Quicksort: Shared Address Space Formulation

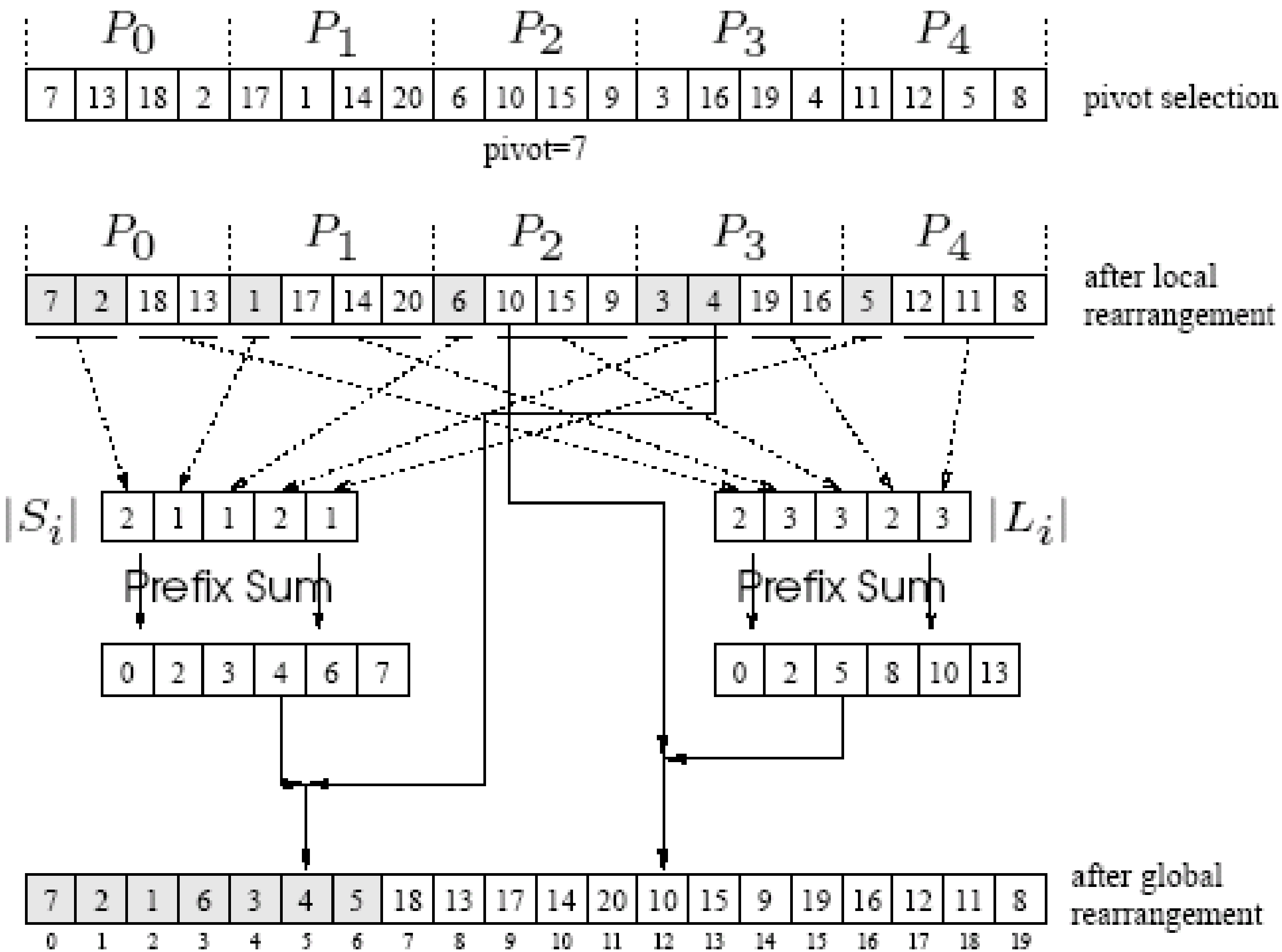
Remaining problem: global reorganization (merging) of local lists to form *L* and *U*.

- The problem is one of determining the right location for each element in the merged list.
- Each processor computes the number of elements locally less than and greater than pivot.
- It computes two *sum-scans* (also called *prefix sum*) to determine the starting location for its elements in the merged *L* and *U* lists.
- Once it knows the starting locations, it can write its elements safely.

Scan operation



- *Parallel prefix sum*: every node got sum of previous nodes + itself



Efficient global rearrangement of the array.

Parallelizing Quicksort: Shared Address Space Formulation

- The parallel time depends on the split and merge time, and the quality of the pivot.
- The latter is an issue independent of parallelism, so we focus on the first aspect, assuming ideal pivot selection.
- One iteration has four steps: (i) determine and broadcast the pivot; (ii) locally rearrange the array assigned to each process; (iii) determine the locations in the globally rearranged array that the local elements will go to; and (iv) perform the global rearrangement.
  - The first step takes time $\Theta(\log p)$, the second, $\Theta(n/p)$, the third, $\Theta(\log p)$, and the fourth, $\Theta(n/p)$.
  - The overall complexity of splitting an n-element array is $\Theta(n/p) + \Theta(\log p)$.

Parallelizing Quicksort: Shared Address Space Formulation

- The process recurses until there are p lists, at which point, the lists are sorted locally.
- Therefore, the total parallel time is:

$$T_P = \overbrace{\Theta\left(\frac{n}{p}\log\frac{n}{p}\right)}^{\text{local sort}} + \overbrace{\Theta\left(\frac{n}{p}\log p\right) + \Theta(\log^2 p)}^{\text{array splits}}. \qquad (4)$$

Useful work
(the same as sequential algorithm)

Overhead (neglectable for large n)

Alternative: PRAM Formulation

- We assume a **CRCW** (concurrent read, concurrent write) PRAM with concurrent writes resulting in an *arbitrary write succeeding* (!!).
- The formulation works by creating pools of processors. Every processor is assigned to the same pool initially and has one element.
- Each processor attempts to write its element to a common location (for the pool).
- Each processor tries to read back the location. If the value read back is greater than the processor's value, it assigns itself to the `left' pool, else, it assigns itself to the `right' pool.
- Each pool performs this operation recursively, in lockstep.
- Note that the algorithm generates a tree of pivots. The depth of the tree is the expected parallel runtime. The average value is $O(\log n)$.

Parallel Quicksort: PRAM Formulation

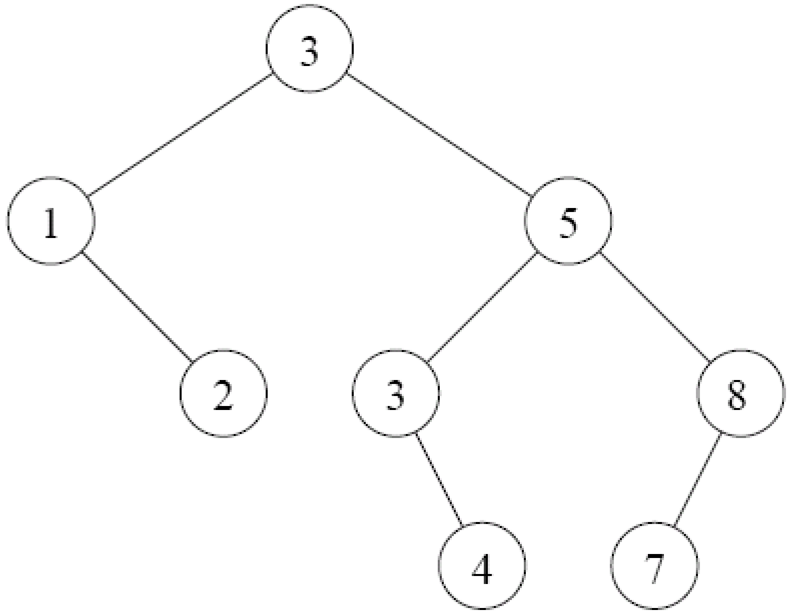
Each thread has 1 value, which will be arranged in a sorted tree

```
while(true){
    write value to pool      barrier
    read pivot from pool
    if (pivot == value)
        break
    else if (pivot < value)
        pool = pool->left
    else
        pool = pool->right
}
```

Performed by all threads in lock-step

⇒ GPU:
Within warps OK, otherwise barrier

Parallel Quicksort: PRAM Formulation



A binary tree generated by the execution of the quicksort algorithm. Each level of the tree represents a different array-partitioning iteration.

If pivot selection is optimal, then the height of the tree is $\Theta(\log n)$, which is also the number of iterations. **Which is almost the ideal speedup!** Overhead = pivot selection.

(a)

1	2	3	4	5	6	7	8
33	21	13	54	82	33	40	72

(b) root = 4

(c)

	1	2	3	4	5	6	7	8
leftchild				1				
rightchild				5				

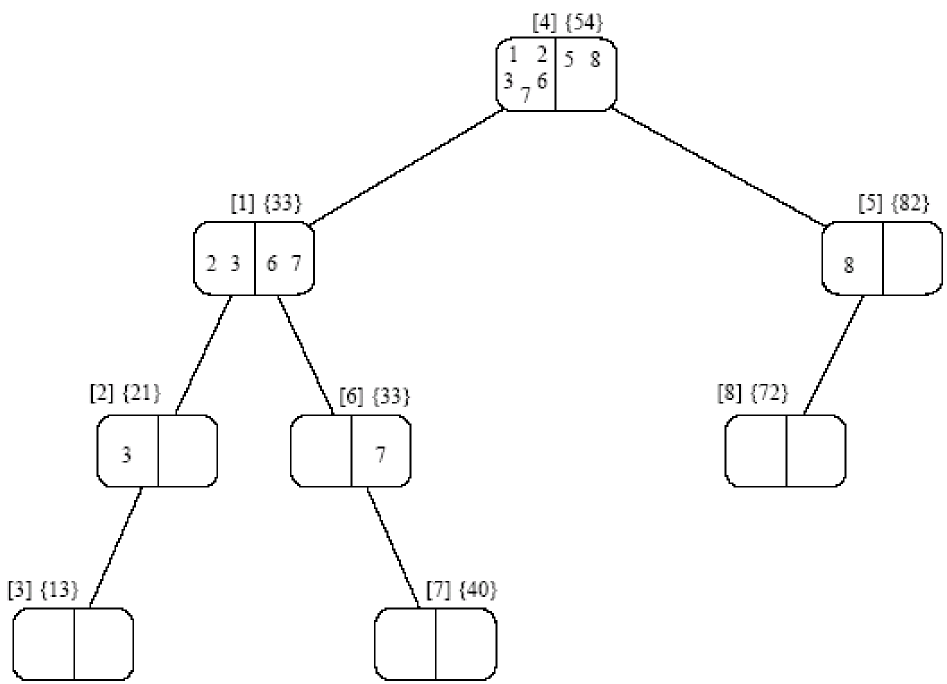
(d)

	1	2	3	4	5	6	7	8
leftchild	2			1	8			
rightchild	6			5				

(e)

	1	2	3	4	5	6	7	8
leftchild	2	3		1	8			
rightchild	6			5		7		

(f)



The execution of the PRAM algorithm on the array shown in (a).

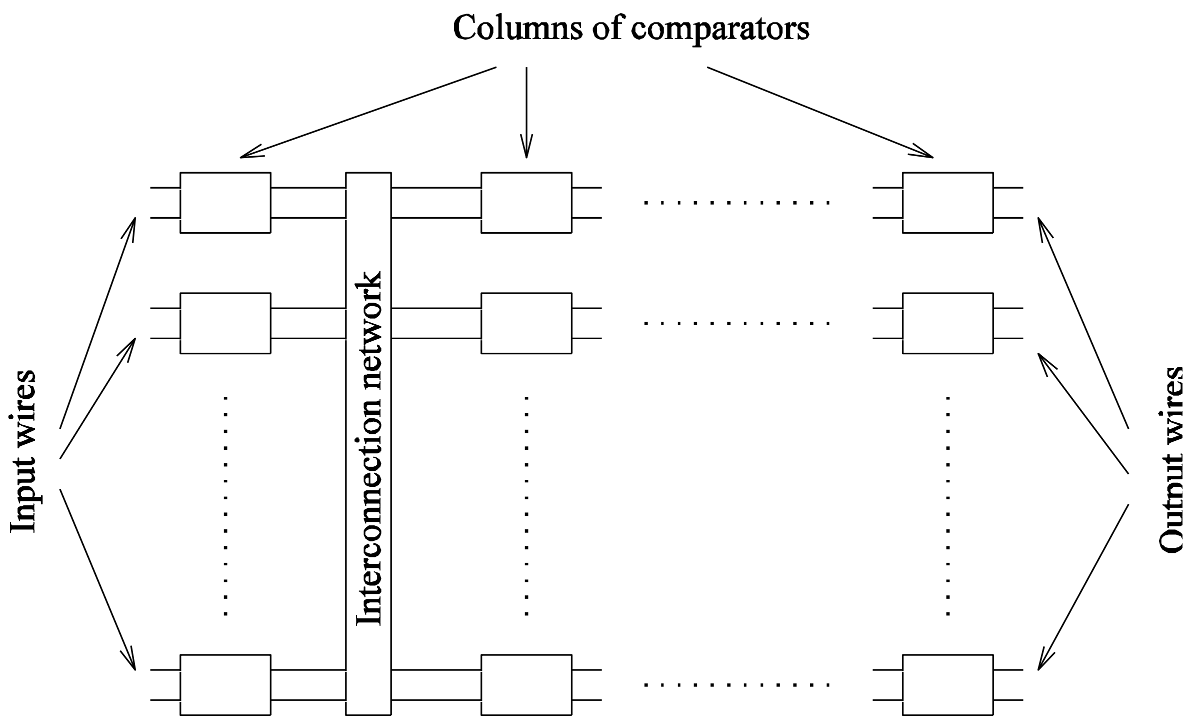
Overview

1. Parallel sort – distributed memory
2. Parallel sort – shared memory
3. **Sorting Networks**
   - A. Odd-even
   - B. Bitonic
4. Parallel sort - GPU

Mission

- Digital circuit that transforms an unsorted list (input) into a sorted list (output)
- Idea: parallel processing! By putting components in parallel *(width)!!*
- So: runtime is determined by *depth*
- Goal: minimal depth

Sorting Networks

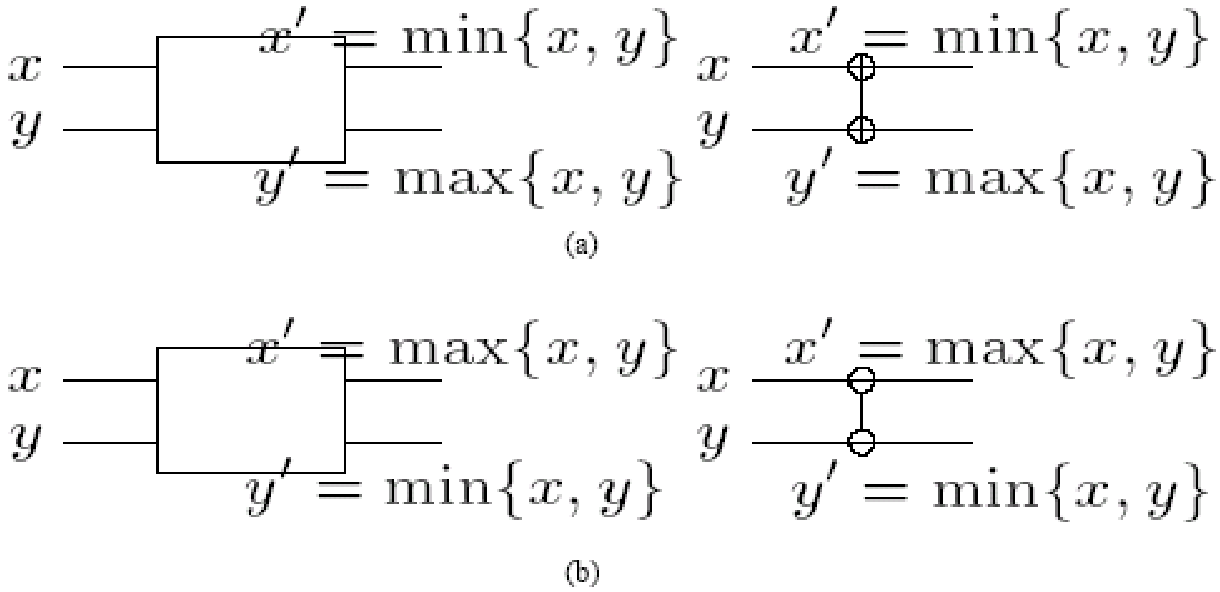


A typical sorting network. Every sorting network is made up of a series of columns, and each column contains a number of comparators connected in parallel.

Sorting Networks

- Networks of comparators designed specifically for sorting (time < $\Theta(n \log n)$).
- Specific-designed parallel system.
- A comparator is a device with two inputs x and y and two outputs x' and y'. For an *increasing comparator*, $x' = \min\{x,y\}$ and $y' = \max\{x,y\}$; and vice-versa for a *decreasing comparator*.
- We denote an increasing comparator by ⊕ and a decreasing comparator by Ө.
- The speed of the network is proportional to its depth.

Basic component: Comparators



A schematic representation of comparators: (a) an increasing comparator, and (b) a decreasing comparator.

KUMAR p. 416

Best algorithm to hardwire?

- Can we sort *n* elements in time *O(log n)*?
  - = quicksort performance
- Quicksort not possible: communication paths are not fixed
- Best: using *O(n.log n)* comparators, but with a quite large constant (many thousands)
  - Not practical
- **Bitonic sort** and **odd-even sort**: sort *n* elements in time $O(\log^2 n)$

Overview

1. Parallel sort – distributed memory

2. Parallel sort – shared memory

3. **Sorting Networks**

    A. **Odd-even**

    B. Bitonic

4. Parallel sort - GPU

Bubble Sort and its Variants

The sequential bubble sort algorithm compares and exchanges adjacent elements in the sequence to be sorted:

```
procedure BUBBLE_SORT(n)
begin
    for i := n − 1 downto 1 do
        for j := 1 to i do
            compare-exchange(a_j, a_{j+1});
end BUBBLE_SORT
```

Sequential bubble sort algorithm.

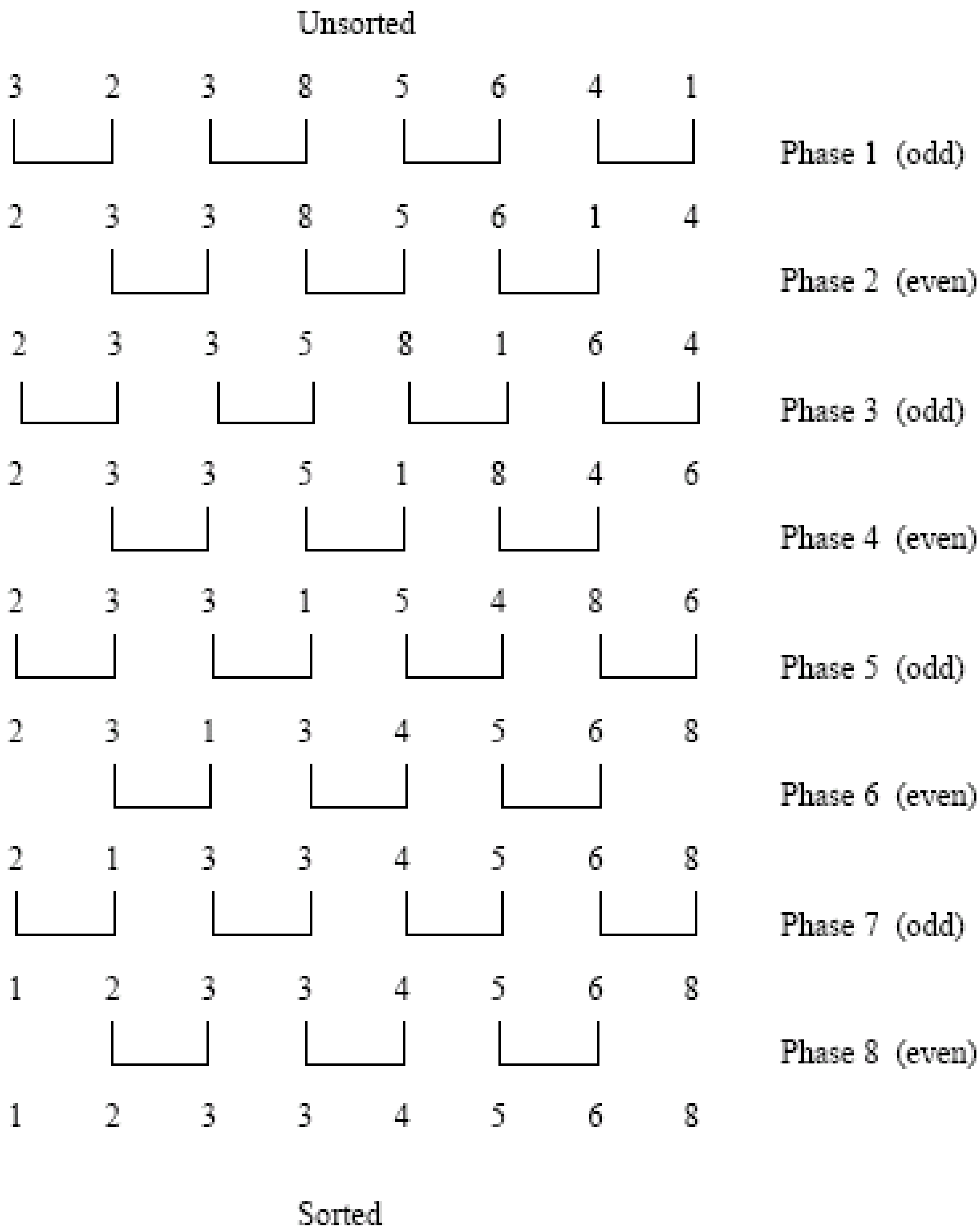
Bubble Sort and its Variants

- The complexity of bubble sort is $\Theta(n^2)$.
- Bubble sort is difficult to parallelize since the algorithm has no concurrency.
- A simple variant, though, uncovers the concurrency.
  - Complexity is lower than quicksort, but parallelization is more efficient

Odd-Even Transposition

```
procedure ODD-EVEN(n)
begin
    for i := 1 to n do
    begin
        if i is odd then
            for j := 0 to n/2 - 1 do
                compare-exchange(a_{2j+1}, a_{2j+2});
        if i is even then
            for j := 1 to n/2 - 1 do
                compare-exchange(a_{2j}, a_{2j+1});
    end for
end ODD-EVEN
```

Sequential odd-even transposition sort algorithm.

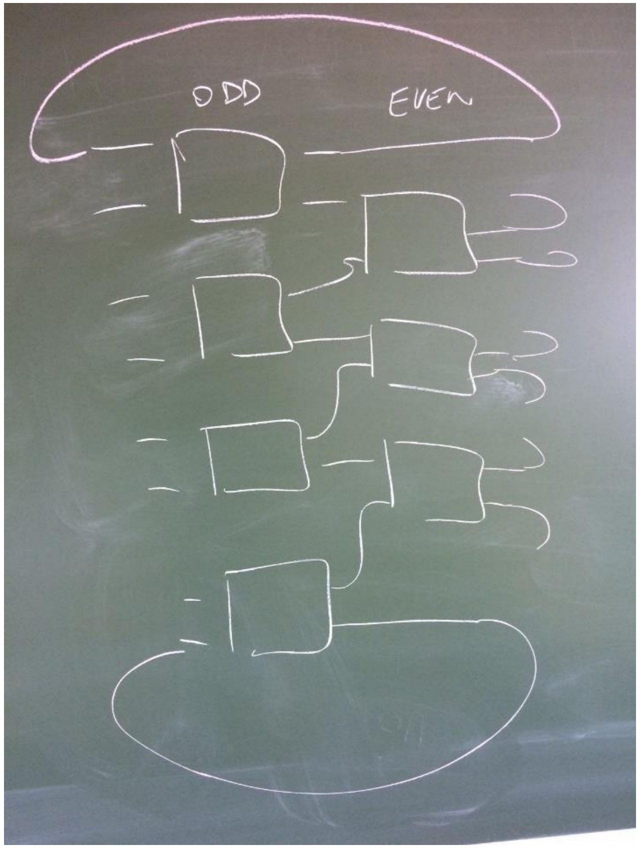


Sorting $n = 8$ elements, using the odd-even transposition sort algorithm. During each phase, $n = 8$ elements are compared.

Odd-Even Transposition

- After *n* phases of odd-even exchanges, the sequence is sorted.
- Each phase of the algorithm (either odd or even) requires $\Theta(n)$ comparisons.
- Serial complexity is $\Theta(n^2)$.
- Parallel version can be implemented by 1 network which is used iteratively!
- *Conclusion*: very simple, but not the fastest

Implementation with 1 network



Use wraparound links
to iterate over both stages

Parallel Odd-Even Transposition

- Consider the one item per processor case.
- There are n iterations, in each iteration, each processor does one compare-exchange.
- The parallel run time of this formulation is $\Theta(n)$.
- This is cost optimal with respect to the base serial algorithm but not to the optimal one ($\Theta(n \log n)$).

```
procedure ODD-EVEN_PAR(n)
begin
    id := process's label
    for i := 1 to n do
    begin
        if i is odd then
            if id is odd then
                compare-exchange_min(id + 1);
            else
                compare-exchange_max(id − 1);
        if i is even then
            if id is even then
                compare-exchange_min(id + 1);
            else
                compare-exchange_max(id − 1);
    end for
end ODD-EVEN_PAR
```

Parallel formulation of odd-even transposition.

Parallel Odd-Even Transposition

- Consider a block of *n/p* elements per processor.
- The first step is a local sort.
- In each subsequent step of *p* steps, the compare exchange operation is replaced by the compare split operation *(n/p* comparisons*).*
- The parallel run time of the formulation is

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

Overview

1. Parallel sort – distributed memory
2. Parallel sort – shared memory
3. **Sorting Networks**
   - A. Odd-even
   - **B. Bitonic**
4. Parallel sort - GPU

Also in PPP118

Sorting Networks: Bitonic Sort

- A bitonic sorting network sorts n elements in $\Theta(\log^2 n)$ time.
- A **bitonic sequence** has two *tones* - increasing and decreasing, or vice versa..
- $\langle 1,2,4,7,6,0 \rangle$ is a bitonic sequence, because it first increases and then decreases.
  - *Not important here*: Any cyclic rotation of a two-tone sequence is also considered bitonic. $\langle 8,9,2,1,0,4 \rangle$ is another bitonic sequence, because it is a cyclic shift of $\langle 0,4,8,9,2,1 \rangle$.

➡ The kernel of the network is the rearrangement of a bitonic sequence into a sorted sequence.

Sorting Networks: Bitonic Sort

- Let $s = \langle a_0, a_1, \ldots, a_{n-1} \rangle$ be a bitonic sequence such that $a_0 \leq a_1 \leq \cdots \leq a_{n/2-1}$ and $a_{n/2} \geq a_{n/2+1} \geq \cdots \geq a_{n-1}$.

- Consider the following subsequences of s:

$$s_1 = \langle \min\{a_0, a_{n/2}\}, \min\{a_1, a_{n/2+1}\}, \ldots, \min\{a_{n/2-1}, a_{n-1}\} \rangle$$
$$s_2 = \langle \max\{a_0, a_{n/2}\}, \max\{a_1, a_{n/2+1}\}, \ldots, \max\{a_{n/2-1}, a_{n-1}\} \rangle \tag{1}$$

- s_1 and s_2 are both bitonic and each element of s_1 is less than every element in s_2.

- We can apply the procedure recursively on s_1 and s_2 to get the sorted sequence.

Bitonic sort’s basic merge component

Basic operation: change a bitonic array into a sorted array.

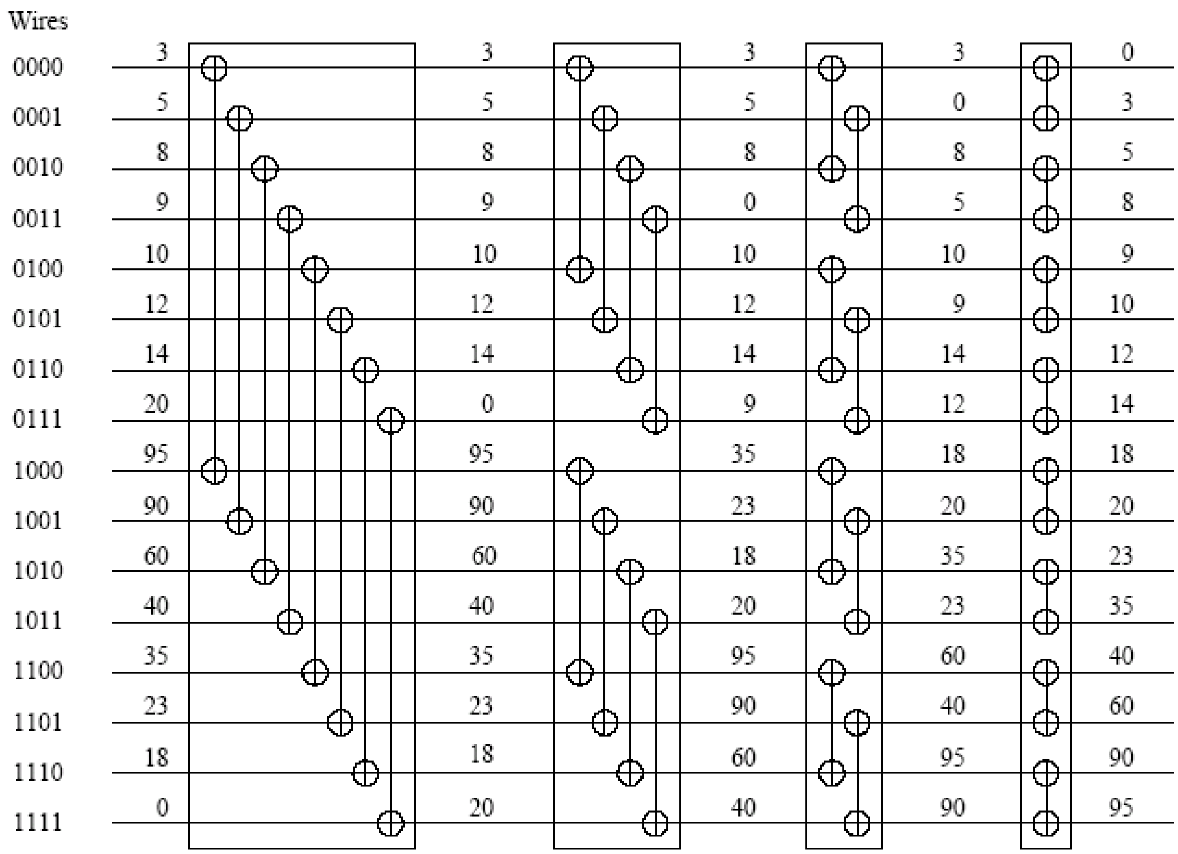
For 16 elements this can be done in 4 steps.

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

Merging a 16-element bitonic sequence through a series of log 16 **bitonic splits**.

The complete network will be based on this component.

- We can easily build a sorting network to implement this bitonic merge algorithm.
- Such a network is called a ***bitonic merging network***.
- The network contains $\log n$ columns. Each column contains $n/2$ comparators and performs one step of the bitonic merge.
- We denote a bitonic merging network with n inputs by **⊕BM[n]**.
- Replacing the ⊕ comparators by ⊖ comparators results in a decreasing output sequence; such a network is denoted by **⊖BM[n]**.



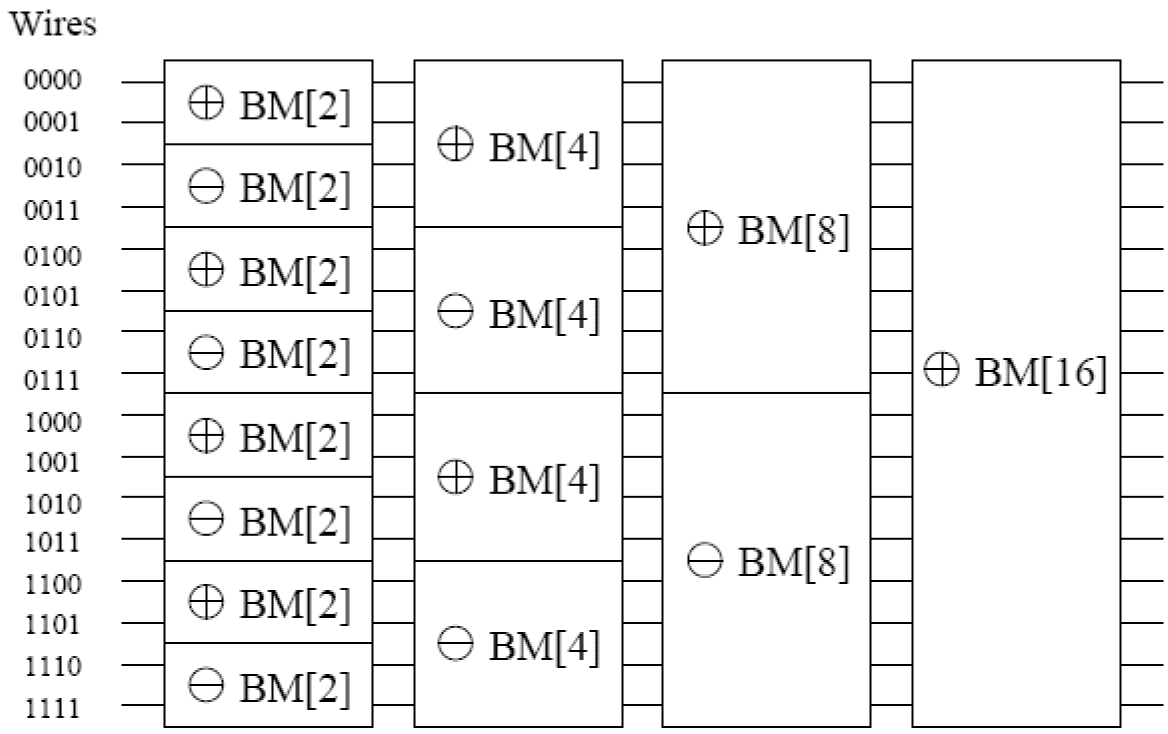
A bitonic merging network for $n = 16$. The input wires are numbered $0,1,\ldots, n - 1$, and the binary representation of these numbers is shown. Each column of comparators is drawn separately; the entire figure represents a $\oplus$BM[16] **bitonic merging network**. The network takes a bitonic sequence and outputs it in sorted order.

Sorting Networks: Bitonic Sort

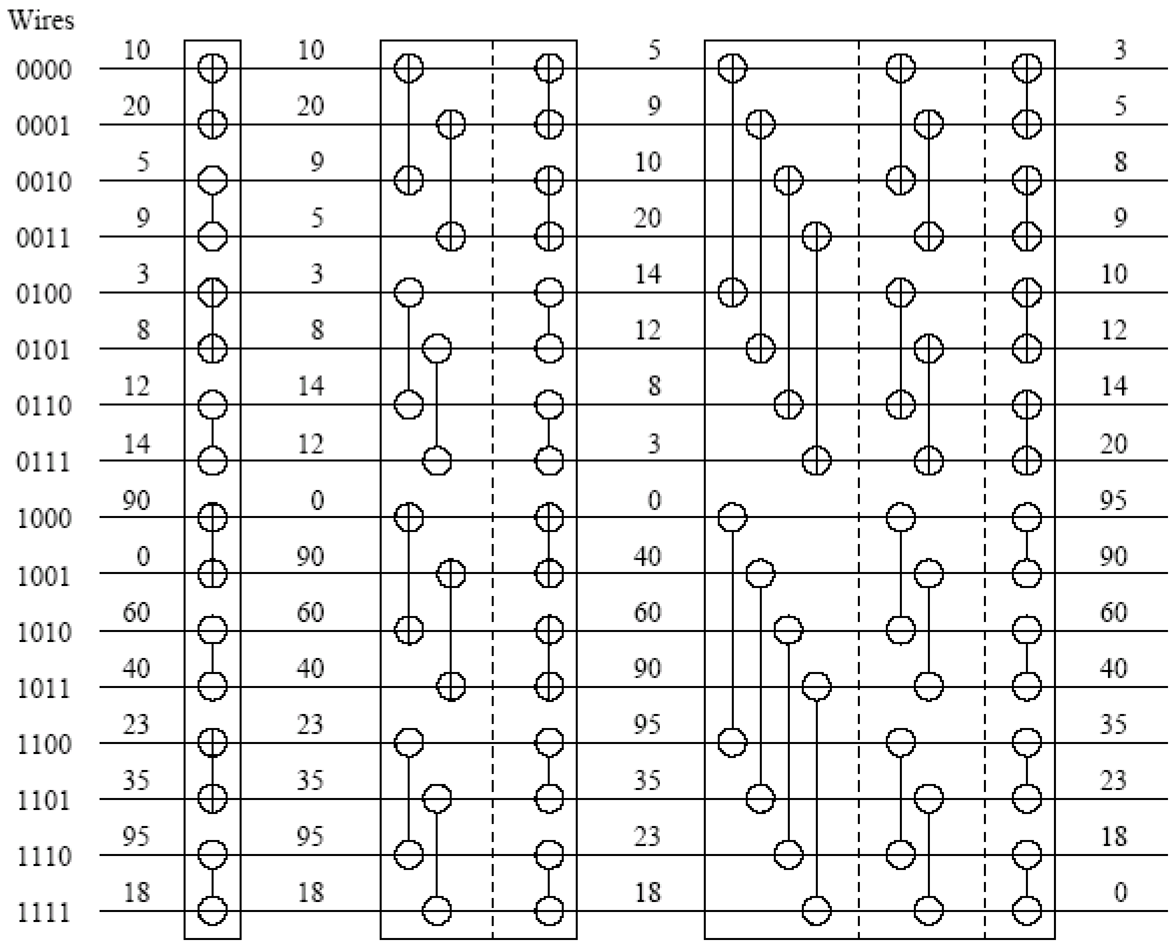
How do we sort an unsorted sequence using a bitonic merge?

- We must first build a single bitonic sequence from the given sequence.
- A sequence of length 2 is a bitonic sequence.
- A bitonic sequence of length 4 can be built by sorting the first two elements using $\oplus BM[2]$ and next two, using $\ominus BM[2]$.

- This process can be repeated to generate larger bitonic sequences.



A schematic representation of a network that converts an input sequence into a bitonic sequence. In this example, ⊕BM[k] and ⊖BM[k] denote bitonic merging networks of input size k that use ⊕ and ⊖ comparators, respectively. The last merging network (⊕BM[16]) sorts the input. In this example, $n = 16$.



The comparator network that transforms an input sequence of 16 unordered numbers into a bitonic sequence.

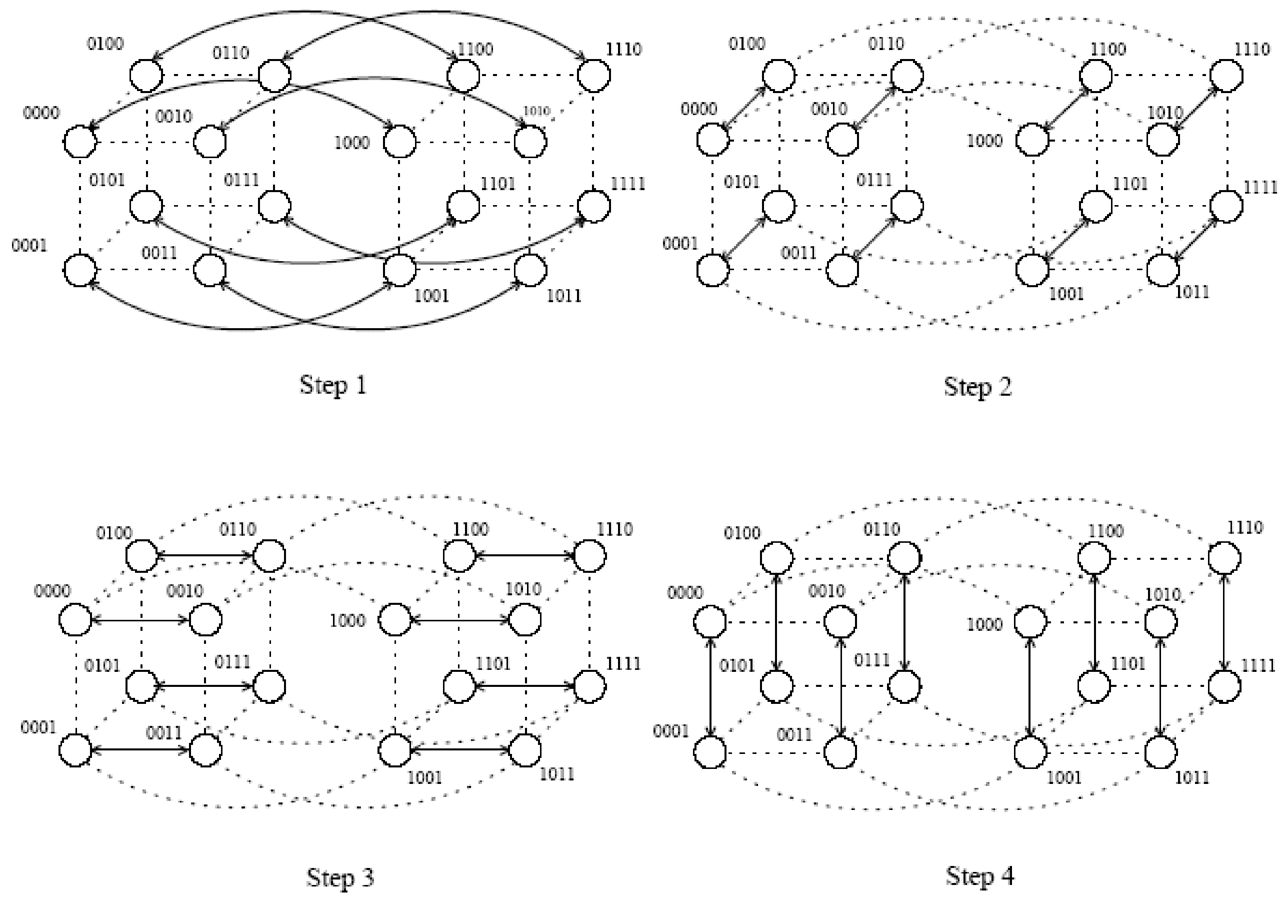
Sorting Networks: Bitonic Sort

- The depth of the network is $\Theta(\log^2 n)$.
  - $1+2+3+\ldots+\log n = (1+\log n).\log n/2$
- Each stage of the network contains $n/2$ comparators. A serial implementation of the network would have complexity $\Theta(n\log^2 n)$.

Mapping Bitonic Sort to Hypercubes

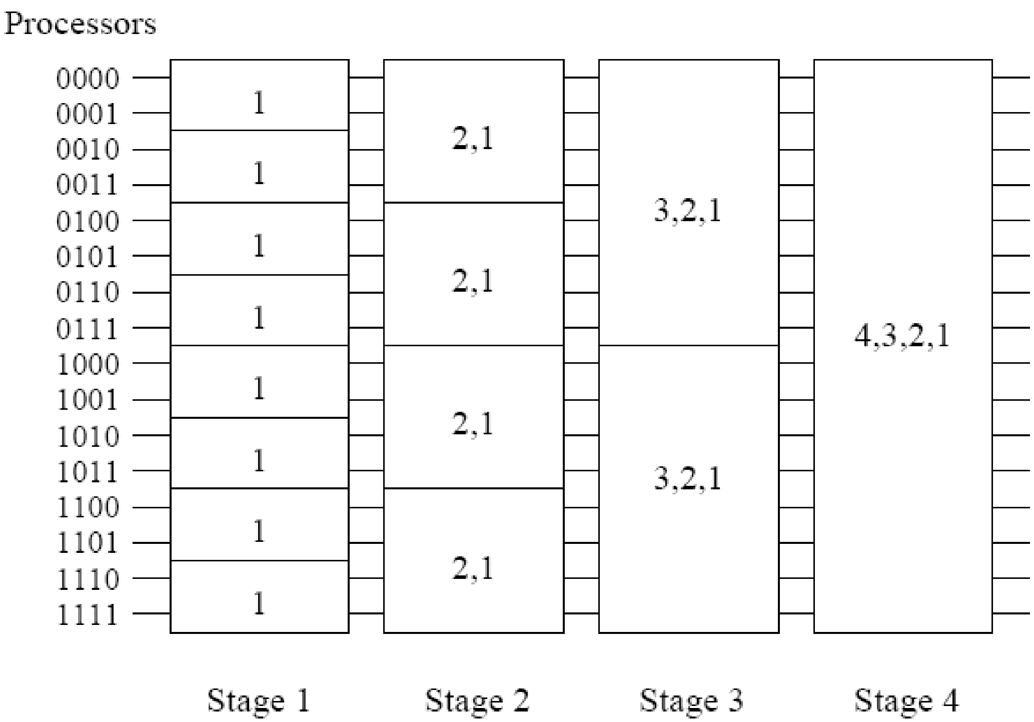
- Map on a general-purpose parallel computer.
- Consider the case of one item per processor. The question becomes one of how the wires in the bitonic network should be mapped to the hypercube interconnect.
- Note from our earlier examples that the compare-exchange operation is performed between two wires only if their labels *differ in exactly one bit*!

➡ a direct mapping of wires to processors; all communication is nearest neighbor!



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 Θ(1) time, the parallel time is

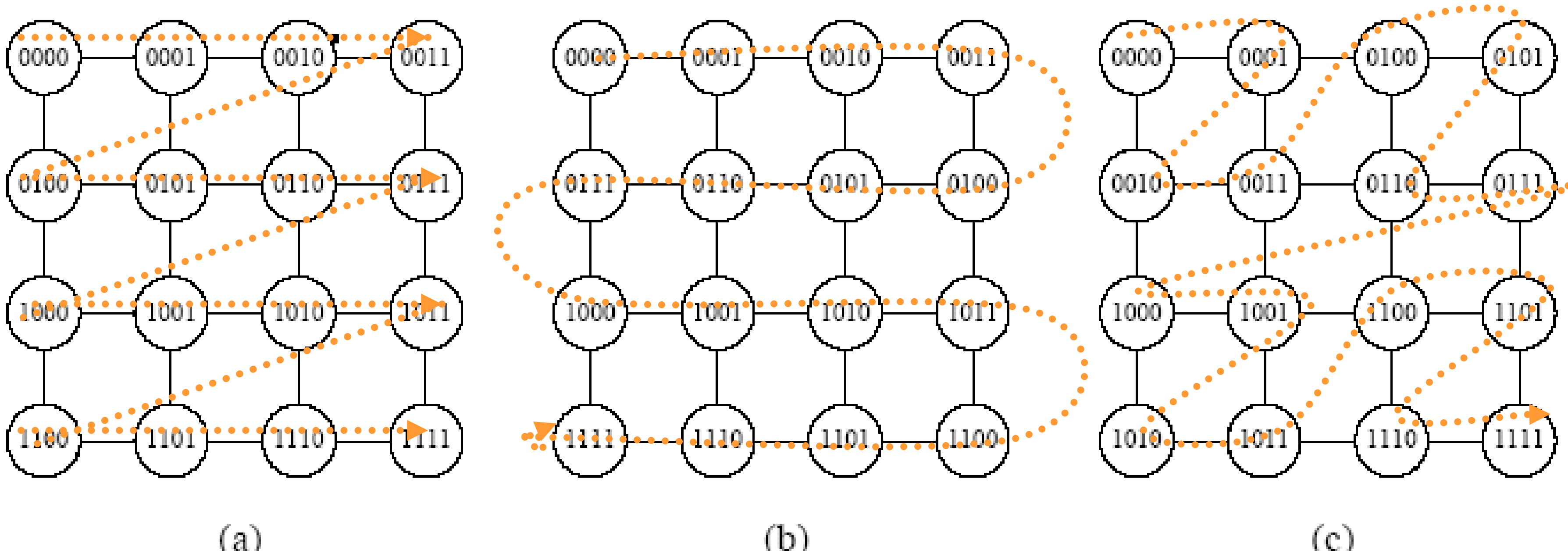
$$T_p = \Theta(\log^2 n) \qquad (2)$$

- This algorithm is cost optimal w.r.t. its serial counterpart, but not w.r.t. the best sorting algorithm ($\Theta(n \log n)$).

Mapping Bitonic Sort to Meshes

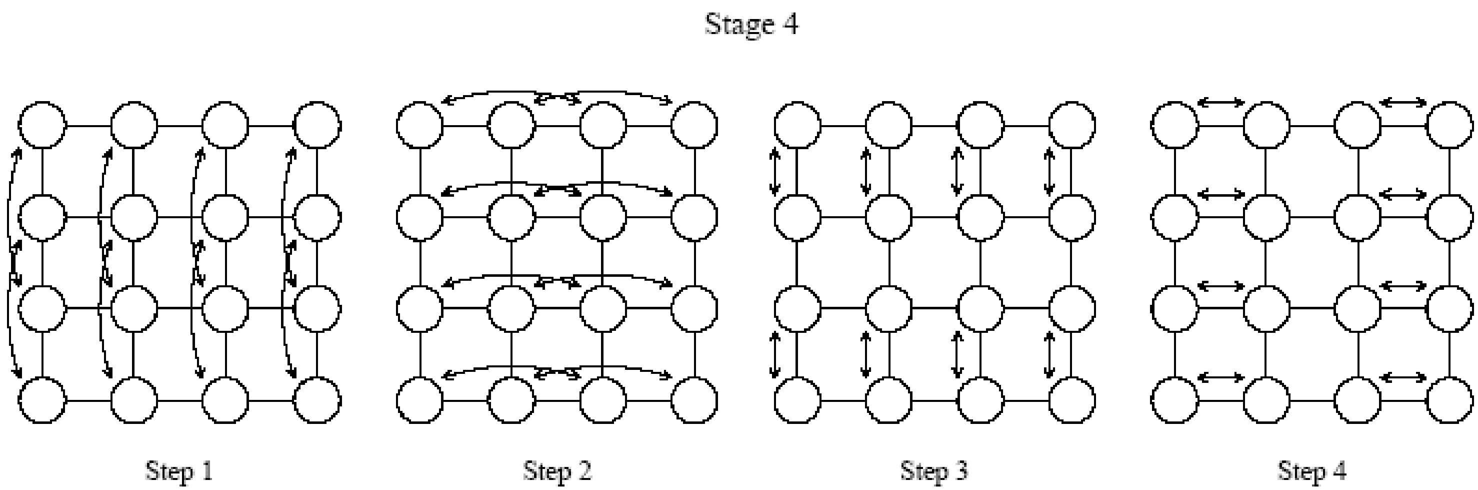
- The connectivity of a mesh is lower than that of a hypercube, so we must expect some overhead in this mapping.
- Consider the row-major shuffled mapping of wires to processors.

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:

$$\sum_{i=1}^{\log n} \sum_{j=1}^{i} 2^{\lfloor (j-1)/2 \rfloor} \approx 7\sqrt{n}, \text{or } \Theta(\sqrt{n})$$

- The total computation performed by each process is $\Theta(\log^2 n)$.
- The parallel runtime is: $T_P = \overbrace{\Theta(\log^2 n)}^{\text{comparisons}} + \overbrace{\Theta(\sqrt{n})}^{\text{communication}}.$

- This is optimal for the mesh, but not cost optimal.

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 => $\mathbf{\Theta(\log^2 p)}$.

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

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

Block of Elements Per Processor: Mesh

- The parallel runtime in this case is given by:

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

Overview

1. Parallel sort – distributed memory
2. Parallel sort – shared memory
3. Sorting Networks
   A. Odd-even
   B. Bitonic
4. **Parallel sort - GPU**

Which algorithms on GPU?

- Quicksort: shared-memory formulation?
- Mergesort? 
- PRAM formulation 
- Odd-even transposition 
- Bitonic sort