Lecture 3: Sorting
Sorting

• Arranging an unordered collection of elements into monotonically increasing (or decreasing) order.

• \( S = \langle a_1, a_2, \ldots, a_n \rangle \) a sequence of \( n \) elements in arbitrary order

After sorting: \( S \) is transformed into a permutation of \( S \):

\[
S' = \langle a'_1, a'_2, \ldots, a'_n \rangle
\]

such that \( a'_i \leq a'_j \) for \( 1 \leq i \leq j \leq n \)

• **Internal vs. external sorting:**
  – Internal sorting: elements to be sorted fit into the processor’s main memory.
  – External sorting: huge number of elements stored into auxiliary storage (hard disks).
Comparison-based vs. Noncomparison-based Sorting

- **Comparison-based sorting:**
  - Repeatedly compare pairs of elements and if they are out of order exchange them
    => compare-exchange operation
  - *Theorem:* The lower bound of any comparison-based sorting algorithm is $\Omega(n \log n)$.
    *i.e. no comparison-based algorithm can possibly be faster than $\Omega(n \log n)$ in the worst case.*

- **Noncomparison-based sorting:**
  - Use certain known properties of the elements (e.g. distribution)
  - *Lower bound:* $\Omega(n)$
Issues in Parallel Sorting

- Where the input and output sequences are stored?
  - Stored on only one process.
  - Distributed among the processes => assumed here
- How comparison are performed?

One element per process:
Compare-Split: More than One Element Per Process

- Each process is assigned a block of $n/p$ elements.
- Block $A_i$ assigned to $P_i$
- $A_i \leq A_j$ if every element of $A_i$ is less than or equal to every element in $A_j$.
- After sorting each $P_i$ holds a set $A'_i$ such that $A'_i \leq A'_j$ for $i \leq j$
- Compare-split operation:
  (assume blocks already sorted at each process)
  1. Each process sends its block to the other process.
  2. Each process merges the two sorted blocks and retains only the appropriate half of the merged block.

Communication cost: $t_s + t_w (n/p)$
Total time: $\Theta(n/p)$
Compare-Split

Step 1

Step 2

Step 3

Step 4
Sorting Networks

Comparators:

(a) Increasing comparator

(b) Decreasing comparator
Sorting Networks

*Depth of a network* = number of columns
The speed of a network is determined by its depth.
Bitonic Sort

• **Bitonic sequence**: a sequence of elements 
  \(<a_0, a_1, ..., a_{n-1}>\) with the property that either:
  1. There exists \(i, 0 \leq i \leq n-1\), such that \(<a_0, a_1, ..., a_i>\) is monotonically increasing and \(<a_{i+1}, a_{i+2}, ..., a_{n-1}>\) is monotonically decreasing, or
  2. There exists a cyclic shift of indices so that (1) is satisfied.

• **Examples**:
  
  \(<1, 2, 4, 7, 6, 0>\)
  \(<8, 9, 2, 1, 0, 4>\) cyclic shift of \(<0, 4, 8, 9, 2, 1>\)
Bitonic Split

Let \( s = <a_0, a_1, \ldots, a_{n-1}> \) a bitonic sequence such that

\[ a_0 \leq a_1 \leq \ldots \leq a_{n/2-1} \]

and

\[ a_{n/2} \geq a_{n/2+1} \geq \ldots \geq a_{n-1} \]

**Bitonic split:**

\[ s_1 = <\min\{a_0, a_{n/2}\}, \min\{a_1, a_{n/2+1}\}, \ldots, \min\{a_{n/2-1}, a_{n-1}\}> \]

\[ s_2 = <\max\{a_0, a_{n/2}\}, \max\{a_1, a_{n/2+1}\}, \ldots, \max\{a_{n/2-1}, a_{n-1}\}> \]

There exists \( b_i \) from \( s_1 \) such that all the elements before \( b_i \) are from the increasing part of \( s \) and all the elements after \( b_i \) are from the decreasing part of \( s \).

There exists \( b'_i \) from \( s_2 \) such that all the elements before \( b'_i \) are from the decreasing part of \( s \) and all the elements after \( b'_i \) are from the increasing part of \( s \).

\[ \Rightarrow s_1 \text{ and } s_2 \text{ are bitonic} \]

Every element of \( s_1 \) is smaller than every element of \( s_2 \)
Bitonic Split

S_2

S_1

a_0 = b_2

a_6

a_7

a_8 = b'_2

a_1

a_2 = b_2

a_3

a_4

a_9

a_{10}

a_{11}

a_5

11
Bitonic Merge

- Reduce the problem of rearranging a bitonic sequence of size $n$ to that of rearranging two smaller bitonic sequences and concatenating the result.
- *Bitonic merge:* Apply recursively the procedure until the length of the sequences is 1
  \[ \Rightarrow \text{sorted sequence} \]
- Takes $\log n$ bitonic splits.
- Can be implemented on a network of comparators \[ \Rightarrow \text{bitonic merging network} \]
## Bitonic Merge: Example

<table>
<thead>
<tr>
<th>Original sequence</th>
<th>3</th>
<th>5</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>12</th>
<th>14</th>
<th>20</th>
<th>95</th>
<th>90</th>
<th>60</th>
<th>40</th>
<th>35</th>
<th>23</th>
<th>18</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1st Split</td>
<td>3</td>
<td>5</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>12</td>
<td>14</td>
<td>0</td>
<td>95</td>
<td>90</td>
<td>60</td>
<td>40</td>
<td>35</td>
<td>23</td>
<td>18</td>
<td>20</td>
</tr>
<tr>
<td>2nd Split</td>
<td>3</td>
<td>5</td>
<td>8</td>
<td>0</td>
<td>10</td>
<td>12</td>
<td>14</td>
<td>9</td>
<td>35</td>
<td>23</td>
<td>18</td>
<td>20</td>
<td>95</td>
<td>90</td>
<td>60</td>
<td>40</td>
</tr>
<tr>
<td>3rd Split</td>
<td>3</td>
<td>0</td>
<td>8</td>
<td>5</td>
<td>10</td>
<td>9</td>
<td>14</td>
<td>12</td>
<td>18</td>
<td>20</td>
<td>35</td>
<td>23</td>
<td>60</td>
<td>40</td>
<td>95</td>
<td>90</td>
</tr>
<tr>
<td>4th Split</td>
<td>0</td>
<td>3</td>
<td>5</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>12</td>
<td>14</td>
<td>18</td>
<td>20</td>
<td>23</td>
<td>35</td>
<td>40</td>
<td>60</td>
<td>90</td>
<td>95</td>
</tr>
</tbody>
</table>
Bitonic Merging Network

log n columns of n/2 comparators
Bitonic Sorting Network
### Bitonic Sorting Network

#### First three stages detailed

<table>
<thead>
<tr>
<th>Wires</th>
<th>20</th>
<th>20</th>
<th>9</th>
<th>9</th>
<th>5</th>
<th>14</th>
<th>12</th>
<th>12</th>
<th>14</th>
<th>3</th>
<th>20</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td></td>
<td></td>
<td>5</td>
<td>10</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>0001</td>
<td></td>
<td></td>
<td>20</td>
<td></td>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>14</td>
<td></td>
</tr>
<tr>
<td>0010</td>
<td></td>
<td>3</td>
<td></td>
<td>14</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>0011</td>
<td></td>
<td>3</td>
<td>8</td>
<td>8</td>
<td>12</td>
<td>3</td>
<td>8</td>
<td>14</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0100</td>
<td></td>
<td>8</td>
<td>8</td>
<td>14</td>
<td>12</td>
<td>3</td>
<td></td>
<td>20</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0101</td>
<td></td>
<td>8</td>
<td>8</td>
<td>14</td>
<td>12</td>
<td>3</td>
<td></td>
<td>20</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0110</td>
<td></td>
<td>8</td>
<td>8</td>
<td>14</td>
<td>12</td>
<td>3</td>
<td></td>
<td>20</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0111</td>
<td></td>
<td>8</td>
<td>8</td>
<td>14</td>
<td>12</td>
<td>3</td>
<td></td>
<td>20</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td></td>
<td>8</td>
<td>8</td>
<td>14</td>
<td>12</td>
<td>3</td>
<td></td>
<td>20</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1001</td>
<td></td>
<td>8</td>
<td>8</td>
<td>14</td>
<td>12</td>
<td>3</td>
<td></td>
<td>20</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1010</td>
<td></td>
<td>8</td>
<td>8</td>
<td>14</td>
<td>12</td>
<td>3</td>
<td></td>
<td>20</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1011</td>
<td></td>
<td>8</td>
<td>8</td>
<td>14</td>
<td>12</td>
<td>3</td>
<td></td>
<td>20</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1100</td>
<td></td>
<td>8</td>
<td>8</td>
<td>14</td>
<td>12</td>
<td>3</td>
<td></td>
<td>20</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1101</td>
<td></td>
<td>8</td>
<td>8</td>
<td>14</td>
<td>12</td>
<td>3</td>
<td></td>
<td>20</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1110</td>
<td></td>
<td>8</td>
<td>8</td>
<td>14</td>
<td>12</td>
<td>3</td>
<td></td>
<td>20</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1111</td>
<td></td>
<td>8</td>
<td>8</td>
<td>14</td>
<td>12</td>
<td>3</td>
<td></td>
<td>20</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Bitonic Sorting Network: Complexity

- **Depth** $d(n)$:
  
  $d(n) = d(n/2) + \log n$
  
  $d(n) = 1 + 2 + \ldots + \log n = (\log^2 n + \log n)/2$
  
  $d(n) = \Theta(\log^2 n)$

- On a serial computer $\Rightarrow \Theta(n \log^2 n)$
Mapping Bitonic Sort on Parallel Computers

- Bitonic sort is communication intensive $\Rightarrow$ we need to take into account the topology of the interconnection network.

- *One element per process:*
  - Each wire of the bitonic sorting network represents a distinct process
  - The compare-exchange operations are performed by $n/2$ pairs of processes

- *Good mapping:* minimizes the distance that the elements travel during a compare-exchange operation.

- *Observation:* wires whose labels differ in the $i$-th least-significant bit perform a compare-exchange operation $(\log n - i + 1)$ times.
Bitonic Sort on a Hypercube

- **Bitonic sort property**: Compare-exchange operations only between wires that differ in only one bit.
- **Hypercube property**: processes whose labels differ in only one bit are neighbors.
- **Optimal mapping**: wire $l$ to process $l$, $l = 0, 1, \ldots, n-1$
- In $i$-th step of the final stage processes communicate along the $(d - (i - 1))$-th dimension.
Bitonic Sort on a Hypercube

Communication during the last stage of bitonic sort
Bitonic Sort on a Hypercube

Communication during the last stage of bitonic sort
Bitonic Sort on a Hypercube

Hypercube dimensions involved in bitonic sort communication
Bitonic Sort on a Hypercube

```
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 $label \neq j^{th}$ bit of $label$ then
                comp_exchange_max(j);
            else
                comp_exchange_min(j);
        end for
    end for
end BITONIC_SORT
```

$T_p = 1 + 2 + \ldots + \log n = (\log^2 n + \log n)/2$

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

$Cost = \Theta(n \log^2 n)$ => cost optimal with respect to the sequential bitonic sort

=> not cost optimal with respect to the best comparison based algorithm
Bitonic Sort on a Mesh

- A mesh has a **lower connectivity** than that of a hypercube
- It is **impossible to map** wires to processes such that each compare-exchange operation occurs only between neighboring processes.
- Map wires such that the most frequent compare-exchange operations occur between neighbors.
Bitonic Sort on a Mesh

row-major

row-major snakelike

row-major shuffled
Bitonic Sort on a Mesh

Stage 4

Step 1
Step 2
Step 3
Step 4

row-major shuffled
Bitonic Sort on a Mesh

- **Property:** Wires that differ in the $i$-th least significant bit are mapped onto mesh processes that are $2^{[(i-1)/2]}$ links away.

- Assuming Store-and-Forward routing:
  \[
  t_{\text{comm}} = \Theta(n^{1/2})
  \]
  \[
  \text{Parallel run time} = \Theta(\log^2 n) + \Theta(n^{1/2})
  \]
  \[
  \text{Cost} = \Theta(n^{1.5}) \Rightarrow \text{not cost optimal}
  \]
More than One Element Per Process

- **Simple way:** \( n/p \) virtual processes on each processor and on hypercube:
  \[ T_p = \Theta((n/p) \log^2 n) \] => not cost-optimal system
- **Another way:** use compare and split operations
- The problem reduces to bitonic sorting of \( p \) blocks using compare-split operations and local sorting of \( n/p \) elements.
- **Total number of steps required:**
  \[ (1 + \log p)(\log p)/2 \]
Bitonic Sort on a Hypercube (p<n)

- Compare-split operation takes $\Theta(n/p)$ computation time and $\Theta(n/p)$ communication time

**Analysis:**
- Initial local sort: $\Theta((n/p) \log (n/p))$
- Comparisons: $\Theta((n/p) \log^2 p)$
- Communication: $\Theta((n/p) \log^2 p)$

- $T_p = \Theta((n/p) \log (n/p)) + \Theta((n/p) \log^2 p)$
- **Cost optimal if**: $\log^2 p = \Theta(\log n)$
  $\Rightarrow p = \Theta(2^{\log^{1/2} n})$

- **Isoefficiency function**: $W = \Theta(p \log p \log^2 p) \Theta(2 \log^2 p \log^2 p)$
  $\Rightarrow$ for large $p$ is worse than polynomial
  $\Rightarrow$ poor scalability
Bitonic Sort on a Mesh (p<n)

- Compare-split operation takes $\Theta(n/p)$ computation time and $\Theta(n/p)$ communication time.
- **Analysis:**
  - Initial local sort: $\Theta((n/p) \log (n/p))$
  - Comparisons: $\Theta((n/p) \log^2 p)$
  - Communication: $\Theta((n/p) p^{1/2})$
- $T_p = \Theta((n/p) \log (n/p)) + \Theta((n/p) \log^2 p) + \Theta(n/p^{1/2})$
- Cost optimal if: $p^{1/2} = \Theta(\log n)$
  \[=> p = \Theta(\log^2 n)\]
- Isoefficiency function: $W = \Theta(2^{p^{1/2}} p^{1/2})$
  \[=> \text{exponential function}\]
  \[=> \text{very poor scalability!}\]
Bubble Sort and its Variants

• **Sequential bubble sort:**
  ▶ (n-1) compare-exchange operations:
    \[(a_1, a_2), (a_2, a_3), ..., (a_{n-1}, a_n)\]
  => it moves the largest element at the end
  ▶ (n-2) compare-exchange operations to the first n-1 elements of the resulting sequence.

After n-1 iterations the sequence is sorted.
Sequential Bubble Sort

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

T_s = \Theta(n^2)

It compares all adjacent pairs in order
=> Inherently sequential
=> Difficult to parallelize!
Odd-Even Transposition

- Sorts $n$ elements in $n$ phases ($n$ even) using $n/2$ compare-exchange operations

- **Odd phase:** compare-exchange between the elements with odd indices and their right neighbors.
  \[
  (a_1, a_2), (a_3, a_4), ..., (a_{n-1}, a_n)
  \]

- **Even phase:** compare-exchange between the elements with odd indices and their right neighbors.
  \[
  (a_2, a_3), (a_4, a_5), ..., (a_{n-2}, a_{n-1})
  \]
Sequential Odd-Even Transposition

$$T_s = \Theta(n^2)$$
Parallel Odd-Even Transposition

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

- Assumes a \( n \) processor ring and one element per process.
- During each phase compare-exchange operations are performed simultaneously \( \Rightarrow \Theta(1) \)
- \( T_p = \Theta(n) \)
- \( Cost = \Theta(n^2) \)
\( \Rightarrow \) not cost-optimal
Parallel Odd-Even Transposition

- More than one element per process ($p < n$):
  - Initial local sort of $n/p$ elements $\Rightarrow \Theta((n/p)\log(n/p))$
  - $p$ phases ($p/2$ odd, $p/2$ even) using compare-split operations.
    Each phase: computation $\Theta(n/p)$, communication $\Theta(n/p)$
    $\Rightarrow$ Total for $p$ phases: computation $\Theta(n)$, communication $\Theta(n)$

- $T_p = \Theta((n/p)\log(n/p)) + \Theta(n) + \Theta(n)$
- Cost optimal when $p = \Theta(\log n)$
- Isoefficiency function: $\Theta(p2^p) \Rightarrow$ exponential
- It is poorly scalable!
Parallel Shellsort

- Compare-exchange between nonadjacent elements.

- **First stage:** processes that are far away from each other in the array compare-split their elements.

- **Second stage:** odd-even transposition in which the odd and even phases are performed only if the blocks on the processes are changing.

- In the first stage the elements are moved close to their final destination => fewer odd and even phases on the second stage.
Parallel Shellsort

first phase
Parallel Shellsort: Analysis

- **Initially**: Local sort of $n/p$ elements: $\Theta((n/p)\log(n/p))$
- **First stage**: each process performs $\log p$ compare-split operations
  Assuming a bisection bandwidth of $\Theta(p)$ then each compare-split operation is $\Theta(n/p)$.
  $\Rightarrow$ first stage takes $\Theta((n/p) \log p)$
- **Second stage**: if $l < p$ odd and even phases are performed each taking $\Theta(n/p)$
  $\Rightarrow$ second stage takes $\Theta(l(n/p))$
- $T_p = \Theta((n/p)\log(n/p)) + \Theta((n/p) \log p) + \Theta(l(n/p))$
- If $l$ small $\Rightarrow$ better than odd-even transposition
- If $l = \Theta(p)$ $\Rightarrow$ similar to odd-even transposition
Sequential Quicksort

• Complexity in the average case: $\Theta(n \log n)$
• Divide-and-conquer algorithm
• *First step* => *divide*
  A[q…r] is partitioned into two nonempty subsequences A[q…s] and A[s+1…r] such that elements of the first are smaller than or equal to each element of the second subsequence.
• *Second step* => *conquer*
  Sort the two subsequences by recursively applying quicksort.
Sequential Quicksort Algorithm

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

Pivot (assumed to be the first element of the subsequence)
Sequential Quicksort: Analysis

- Complexity of partitioning a sequence of size $k$: $\Theta(k)$
- **Worst case**: always partition into a sequence of one element and one of $k-1$ elements
  - $T(n) = T(n-1) + \Theta(n)$
  - $T(n) = \Theta(n^2)$
- **Best case**: always partition into two roughly equal size subsequences
  - $T(n) = 2T(n/2) + \Theta(n)$
  - $T(n) = \Theta(n \log n)$
- **Average case**: equally likely to break the sequence into partitions of size 0 and $n-1$, 1 and $n-2$, etc.
  - $T(n) = (1/n)\sum_{k=0}^{n-1} (T(k) + T(n-1-k)) + \Theta(n)$
  - $T(n) = \Theta(n \log n)$
Parallelizing Quicksort

- *Naïve parallelization*: execute it initially on a single process then assign one of the subproblems to another process and so on.
- Upon termination each process contains one element
- *Major drawback*: one process must partition the initial array of size $n \Rightarrow$ runtime is bounded below by $\Omega(n) \Rightarrow$ not cost optimal
- *Efficient parallelization*: do the partitioning steps in parallel!
Parallelizing Quicksort

First Step

<table>
<thead>
<tr>
<th>$P_0$</th>
<th>$P_1$</th>
<th>$P_2$</th>
<th>$P_3$</th>
<th>$P_4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>7 13 18 2 17 1 14 20 6 10 15 9 3 16 19 4 11 12 5 8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

pivot selection

<table>
<thead>
<tr>
<th>$P_0$</th>
<th>$P_1$</th>
<th>$P_2$</th>
<th>$P_3$</th>
<th>$P_4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>7 2 18 13 1 17 14 20 6 10 15 9 3 4 19 16 5 12 11 8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

after local rearrangement

<table>
<thead>
<tr>
<th>$P_0$</th>
<th>$P_1$</th>
<th>$P_2$</th>
<th>$P_3$</th>
<th>$P_4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>7 2 1 6 3 4 5 18 13 17 14 20 10 15 9 19 16 12 11 8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

after global rearrangement

Second Step

<table>
<thead>
<tr>
<th>$P_0$</th>
<th>$P_1$</th>
<th>$P_2$</th>
<th>$P_3$</th>
<th>$P_4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>7 2 1 6 3 4 5 18 13 17 14 20 10 15 9 19 16 12 11 8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

pivot selection

<table>
<thead>
<tr>
<th>$P_0$</th>
<th>$P_1$</th>
<th>$P_2$</th>
<th>$P_3$</th>
<th>$P_4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 2 7 6 3 4 5 14 13 17 18 20 10 15 9 19 16 12 11 8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

pivot=17

<table>
<thead>
<tr>
<th>$P_0$</th>
<th>$P_1$</th>
<th>$P_2$</th>
<th>$P_3$</th>
<th>$P_4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 2 3 4 5 7 6 14 13 17 10 15 9 16 12 11 8 18 20 19</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

after local rearrangement

<table>
<thead>
<tr>
<th>$P_0$</th>
<th>$P_1$</th>
<th>$P_2$</th>
<th>$P_3$</th>
<th>$P_4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 2 3 4 5 7 6 14 13 17 10 15 9 16 12 11 8 18 20 19</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

after global rearrangement
Parallelizing Quicksort

Third Step

Fourth Step

Solution
Efficient Global Rearrangement

pivot selection

after local rearrangement

after global rearrangement
Parallelizing Quicksort

• Four steps for splitting:
  1. Determine and broadcast the pivot $\Rightarrow \Theta(\log p)$
  2. Locally rearrange the array assigned to each process $\Rightarrow \Theta(n/p)$
  3. Determine the locations in the globally rearranged array that the local elements will go to $\Rightarrow$ two prefix sum operations $\Rightarrow \Theta(\log p)$
  4. Perform the global rearrangement $\Rightarrow \Theta(n/p)$ (lower bound)

• Each of these four steps are executed $\log p$ times $\Rightarrow$

  Total time for splitting: $\Theta((n/p)\log(p)) + \Theta(\log^2 p)$

• The last phase is to sort locally the elements $\Rightarrow$
  $\Theta((n/p)\log(n/p))$

• $T_p = \Theta((n/p)\log(n/p)) + \Theta((n/p)\log p) + \Theta(\log^2 p)$

• Isoefficiency: $\Theta(p \log^2 p)$
Pivot Selection

- The split should be done such that each partition has a non-trivial fraction of the original array.
- *Random:*
  - During the $i$-th split one process in each group randomly selects one of its elements to be the pivot.
  - Bad for the parallel implementation
- *Median:*
  - Assumes a uniform distribution of the elements in the initial array.
  - $n/p$ elements stored at each process form a representative sample of all $n$ elements.
  - Distribution of elements at each process is the same as the overall distribution.
  - The median at each process approximates the overall median.
  - Pivot = local median
- Maintains balanced subsequences throughout the execution of the algorithm.
Bucket Sort

- Sorts an array of \( n \) elements whose values are uniformly distributed over an interval \([a,b]\)
- **Sequential Bucket Sort:**
  - Divide \([a,b]\) into \( m \) equal-sized subintervals => buckets
  - Place the elements in the appropriate bucket => approximately \( n/m \) elements per bucket
  - Sort the elements in each bucket.
- \( T_s = \Theta(n \log(n/m)) \)
- If \( m = \Theta(n) \) => \( T_s = \Theta(n) \) linear time!
Parallel Bucket Sort

- Each process represents a bucket \((m = p)\)
- **Parallel Bucket Sort:**
  - Each process partitions its block into \(p\) processes.
  - Each process sorts its bucket.
- The uniform distribution assumption is not realistic
- If the distribution is not uniform sub-blocks
  => buckets have a significantly different number of elements
  => degrading the performance
Sample Sort

- Does not assume uniform distribution.
- A sample of size $s$ is selected from the $n$ element sequence.
- Sort the sample and select $m-1$ elements (splitters).
- Splitters determine the range of the buckets.
- After the buckets are determined the algorithm proceeds as in bucket sort.
Parallel Sample Sort

Initial element distribution

Local sort & sample selection

Sample combining

Global splitter selection

Final element assignment
Parallel Sample Sort: Analysis

- $p$ processes message-passing computer and $O(p)$ bisection bandwidth.
- Parallel Sample Sort
  - Internal sort => $\Theta((n/p)\log (n/p))$
  - Selection of $p-1$ samples => $\Theta(p)$
  - Send $p-1$ elements to $P_0$ (gather) => $\Theta(p^2)$
  - Internal sort of $p(p-1)$ elements at $P_0$ => $\Theta(p^2 \log p)$
  - Select $p-1$ splitters at $P_0$ => $\Theta(p)$
  - Broadcast $p-1$ splitters => $\Theta(p \log p)$
  - Each process partitions its blocks into $p$ sub-blocks, one for each bucket => $p-1$ binary searches => $\Theta(p \log (n/p))$
  - Each process sends sub-blocks to the appropriate processes (if distribution is close to uniform then size of sub-blocks is approx $n/p^2$ and assume Hypercube) $(t_s + t_w, mp/2)\log p$ => $\Theta((n/p)\log p)$
- $T_p = \Theta((n/p)\log (n/p)) + \Theta(p^2 \log p) + \Theta(p \log (n/p)) + \Theta((n/p)\log p)$
- Isoefficiency function: $\Theta(p^3 \log p)$