Sorting methods

• Classification of sorting algorithms
  – *internal* vs *external*
    • internal: input data set small enough to fit into memory
  – *comparison-based* vs *noncomparison-based*
    • the former algs are based on pairwise comparison and exchange (compare-and-exchange is base operation)
    • The later algs sort by using certain known properties of the elements such as their binary representation or their distribution.
    • lower bound on the sequential complexity is $\Theta(n \log n)$ vs $\Theta(n)$
Basic sorting operations: compare-exchange

- Problem: how to perform compare-exchange on a parallel system with one element per processor
- Solution: send element to other node, then perform comparison
- Running time:
  - $T_{tot} = t_{comp} + t_{comm}$
  - $t_{comm} = t_s + t_w \approx t_s$ (assuming neighboring nodes)
Basic sorting operations: compare-split

- Problem: how to perform compare-exchange on a parallel system with \( n/p \) elements per processor
- Solution: send elements to other node, then merge and retain only half of the elements
- Running time:
  - \( T_{\text{tot}} = t_{\text{comp}} + t_{\text{comm}} \)
  - \( t_{\text{comm}} = t_s + n/p \ t_w \approx n/p \ t_w \) (neighboring nodes, \( n \gg p \))
Bubble sort

- The serial version compares all adjacent pairs in order:
  - \((a_1, a_2), (a_2, a_3), \ldots, (a_{n-1}, a_n)\)
  - iterate \(n\) times
  - complexity: \(\Theta(n^2)\)
- Some modification to base algorithm needed for parallelization
- Odd-Even Transposition: perform compare-exchange on odd elements, then on even elements
  - \((a_1, a_2), (a_3, a_4), \ldots, (a_{n-1}, a_n)\)
  - \((a_2, a_3), (a_4, a_5), \ldots, (a_{n-2}, a_{n-1})\)
  - iterate \(n\) times
  - complexity: \(\Theta(n^2)\)

Procedure BUBBLE_SORT(n)
begin
  for \(i := 1\) to \(n-1\) do
    for \(j := 0\) to \(n-i-1\) do
      compare-exchange\((a_j, a_{j+1})\)
  end
end

Procedure ODD_EVEN(n)
begin
  for \(i := 1\) to \(n\) do
    begin
      if \(i\) is odd then
        for \(j := 1\) 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\((a_{2j}, a_{2j+1})\)
    endfor
end
Odd-Even transposition example
Parallel bubble sort

• Assume ring interconnect
• Simple case: \( p = n \)
  – Running time: \( \Theta(n) \)
    • \( n \) iterations, one compare-exchange per iteration (complexity: \( \Theta(1) \))
  – Cost: \( \Theta(n^2) \)
    • not cost optimal - compare to \( \Theta(n \log n) \)
• General case: \( p < n \)
  – Running time: \( \Theta(n/p \log n/p) + \Theta(n) \)
    • each processor sorts internally its block of \( n/p \) element (for example using quicksort-complexity: \( \Theta(n/p \log n/p) \))
    • \( p \) phases each with
      – \( \Theta(n/p) \) comparisons (to merge blocks)
      – \( \Theta(n/p) \) communication time
  – \( E = 1/(1 - \Theta((\log p)/(\log n)) + \Theta((p)/(\log n)) ) \)
    i.e. cost-optimal when \( p = O(\log n) \)

```
Procedure ODD_EVEN_PAR(n)
begin
  id := processor’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)
      else
        if id is even then
          compare-exchange_min(id + 1)
        else
          compare-exchange_max(id - 1)
    endfor
end
```
Quicksort

- The recursive algorithm consists of four steps (which closely resemble the merge sort):
  - If there are one or less elements in the array to be sorted, return immediately.
  - Pick an element in the array to serve as a "pivot" point. (Usually the left-most element in the array is used.)
  - Split the array into two parts - one with elements larger than the pivot and the other with elements smaller than the pivot.
  - Recursively repeat the algorithm for both halves of the original array.
- Performance is affected by the way the algorithm splits the sequence
  - worst case (1 and k-1 splitting): recurrent relations
    - $T(n) = T(n-1) + \Theta(n) \Rightarrow T(n) = \Theta(n^2)$
  - best case (k/2 and k/2 splitting):
    - $T(n) = 2T(n/2) + \Theta(n) \Rightarrow T(n) = \Theta(n \log n)$
Quicksort example

(a) 3 2 1 5 8 4 3 7
(b) 1 2 3 5 8 4 3 7
(c) 1 2 3 3 4 5 8 7
(d) 1 2 3 3 4 5 7 8
(e) 1 2 3 3 4 5 7 8

Example
Quicksort efficient parallelization

- Drawback of naïve approach: the initial partitioning of \( A[q \ldots r] \) is done by a single processor
  - run time is bounded below by \( O(n) \)
  - cost is \( O(n^2) \) therefore not cost-optimal
- Complexity of quicksort algorithm:
  - \( T(n) = 2T(n/2) + \Theta(n) \Rightarrow \Theta(n\log n) \) (for optimal pivot selection)
    - the term \( \Theta(n) \) is due to the partitioning
    - the same term could become \( \Theta(1) \) if we find a way of parallelizing the partitioning using \( n \) processors
      - will see solutions for PRAM and hypercube
Shared memory machine: PRAM model

- Parallel Random Access Machine (PRAM) is a popular model used in the design of parallel algorithms
  - It assumes a number of processors with a single shared memory
  - Variants based on concurrency of accesses:
    - EREW: Exclusive Read, Exclusive Write
    - CREW: Concurrent Read, Exclusive Write
    - CRCW: Concurrent Read, Concurrent Write
Parallel version on a PRAM (1)

- The execution of the algorithm can be represented with a tree
  - the root is the initial pivot
  - each level represents a different iteration
- If pivot selection is optimal, the height of the tree is $\Theta(\log n)$
- The parallel algorithm proceeds by selecting an initial pivot, then partitioning the array in two parts in parallel
Parallel version on a PRAM (2)

- We will consider a CRCW PRAM
  - concurrent read, concurrent write parallel random access machine
  - when two or more processors write to a common location only one arbitrarily chosen is successful
- The algorithm is based on two shared arrays, \textit{leftchild} and \textit{rightchild}, where all processors write at each iteration
  - the CRCW arbitration mechanism is used to pick the next pivot
  - average depth of the tree is $\Theta(\log n)$, each step takes $\Theta(1)$, thus
  - average complexity is $\Theta(\log n)$
  - average cost is $\Theta(n \log n) \Rightarrow$ cost optimal
Figure 9.17  The execution of the PRAM algorithm on the array shown in (a). The arrays leftchild and rightchild are shown in (c), (d), and (e) as the algorithm progresses. Figure (f) shows the binary tree constructed by the algorithm. Each node is labeled by the process (in square brackets), and the element is stored at that process (in curly brackets). The element is the pivot. In each node, processes with smaller elements than the pivot are grouped on the left side of the node, and those with larger elements are grouped on the right side. These two groups form the two partitions of the original array. For each partition, a pivot element is selected at random from the two groups that form the children of the node.
Sequence of Elements

(a) Split along the third dimension. Partitions the sequence into two big blocks—one smaller and one larger than the pivot.

(b) Split along the second dimension. Partitions each subblock into two smaller subblocks.

(c) Split along the first dimension. The elements are sorted according to the global ordering imposed by the processors’ labels onto the hypercube.

Figure 9.21 The execution of the hypercube formulation of quicksort for $d = 3$. The three splits—one along each communication link—are shown in (a), (b), and (c). The second column represents the partitioning of the $n$-element sequence into subcubes. The arrows between subcubes indicate the movement of larger elements. Each box is marked by the binary representation of the process labels in that subcube. A * denotes that all the binary combinations are included.
Parallel version on hypercube

- This algorithm exploits one property of hypercubes:
  - a $d$-dimensional hypercube can be split in two $(d-1)$-dimensional hypercubes with the corresponding nodes directly connected
  - $n$ elements are distributed on $p = 2^d$ processors ($n/p$ elements per processor)

- At each iteration, pivot is chosen and broadcast to all processors in same hypercube
  - then smaller-than-pivot elements are sent to half hypercube, the larger ones to the other half

- Selection of good pivot is crucial to maintain good load balance
  - a good criterion is to choose the median element of an arbitrarily selected processor in the hypercube (works well with uniform distribution)
procedure HYPERCUBE_QUICKSORT (B, n)
begin
    id := processor's label;
    for i := 1 to d do
        begin
            x := pivot;
            partition B into B_1 and B_2 such that B_1 ≤ x < B_2;
            if i^{th} bit is 0 then
                begin
                    send B_2 to the processor along the i^{th} communication link;
                    C := subsequence received along the i^{th} communication link;
                    B := B_1 U C;
                endif
            else
                begin
                    send B_1 to the processor along the i^{th} communication link;
                    C := subsequence received along the i^{th} communication link;
                    B := B_2 U C;
                endelse
        endfor
    sort B using sequential quicksort;
end HYPERCUBE_QUICKSORT
Hypercube algorithm complexity

• The algorithms performs $d$ iterations, each has three steps
  – pivot selection $\Rightarrow \Theta(1)$ if $p/n$ elements are presorted
  – broadcast of pivot $\Rightarrow \Theta(\log n)$

  – $Tp = \Theta(n/p \log n/p)_{\text{local sort}} + \Theta(n/p \log p)_{\text{comm}} + \Theta(\log^2 p)_{\text{pivot broadcasting}}$

• Efficiency and cost-optimality analysis
  – $E = 1/(1 - \Theta((\log p)/(\log n)) + \Theta((p \log^2 p)/(n \log n)))$
  – cost-optimal if $\Theta((p \log^2 p)/(n \log n)) = O(1)$, i.e. can use up to $p = \Theta(n/\log n)$ processors efficiently
Sorting networks

- Key component is the comparator
- A sorting network is usually built of several columns of comparators
- We will see one example of sorting network:
  - bitonic merging network
Bitonic sequences

• A *bitonic sequence* is a sequence of elements \((a_0, a_1, \ldots, a_n)\) such that:
  - there exist an index \(i\) such that \((a_0, a_1, \ldots, a_i)\) is monotonically increasing, and \((a_{i+1}, \ldots, a_n)\) is monotonically decreasing
  - or there exist a shift of indices such the above is true

• Consider the subsequences:
  - \(s_1\) and \(s_2\) are bitonic, and each element of \(s_1\) is smaller than any element of \(s_2\)
  - the splitting of \(s\) in \(s_1\) and \(s_2\) is called bitonic split
Bitonic merge

- The sorting of a bitonic sequence using bitonic splits is called *bitonic merge*
  - this task can be easily implemented with a network of comparators

<table>
<thead>
<tr>
<th>Original sequence</th>
<th>3 5 8 9 10 12 14 20 95 90 60 40 35 23 18 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1st Split</td>
<td>3 5 8 9 10 12 14 0 95 90 60 40 35 23 18 20</td>
</tr>
<tr>
<td>2nd Split</td>
<td>3 5 8 0 10 12 14 9 35 23 18 20 95 90 60 40</td>
</tr>
<tr>
<td>3rd Split</td>
<td>3 0 8 5 10 9 14 12 18 20 35 23 60 40 95 90</td>
</tr>
<tr>
<td>4th Split</td>
<td>0 3 5 8 9 10 12 14 18 20 23 35 40 60 90 95</td>
</tr>
</tbody>
</table>
- Bitonic merging network of size 16 (denoted as $\oplus\text{BM}[16]$)
Bitonic sorting

- To sort an unordered sequence of elements using the bitonic merge, we first need to convert it into a (unsorted) bitonic sequence
  - note that a pair of elements can be considered a bitonic sequence of length two ...
Figure 9.8 The comparator network that transforms an input sequence of 16 unordered numbers into a bitonic sequence. In contrast to Figure 9.6, the columns of comparators in each bitonic merging network are drawn in a single box, separated by a dashed line.
Figure 9.9 Communication during the last stage of bitonic sort. Each wire is mapped to a hypercube process; each connection represents a compare-exchange between processes.
Figure 9.10  Communication characteristics of bitonic sort on a hypercube. During each stage of the algorithm, processes communicate along the dimensions shown.