Sorting Algorithms

Slides used during lecture of 21/11/2014
(D. Roose)

Adapted from slides by
Ananth Grama, Anshul Gupta, George Karypis, and Vipin Kumar

To accompany the text ``Introduction to Parallel Computing'',
Addison Wesley, 2003.

Topic Overview

- Issues in Sorting on Parallel Computers
- Bubble Sort and its Variants
- Quicksort
- Sorting Networks
- Bucket and Sample Sort
- Other Sorting Algorithms

Sorting: Overview

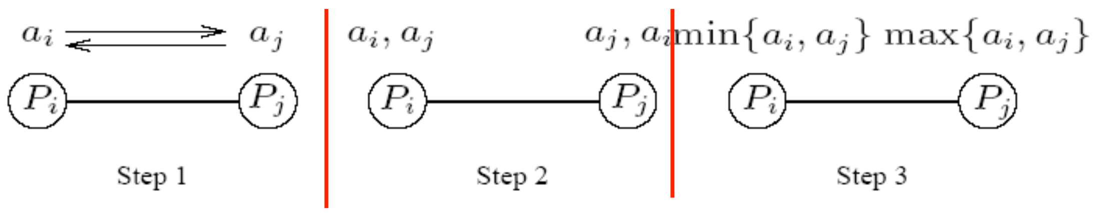
- One of the most commonly used and well-studied kernels.
- Sorting can be *comparison-based* or *non-comparison-based*.
- The fundamental operation of comparison-based sorting is *compare-exchange*.
- The lower bound on any comparison-based sort of n numbers is $O(n \log n)$.
- We focus here on comparison-based sorting algorithms.

Sorting: Basics

What is a parallel sorted sequence?
Where are the input and output lists stored?

- We assume that the input and output lists are distributed.
- The sorted list is partitioned with the property that each partitioned list is sorted and each element in processor P_i's list is less than that in P_j's list if $i < j$.

Sorting: Parallel Compare Exchange Operation

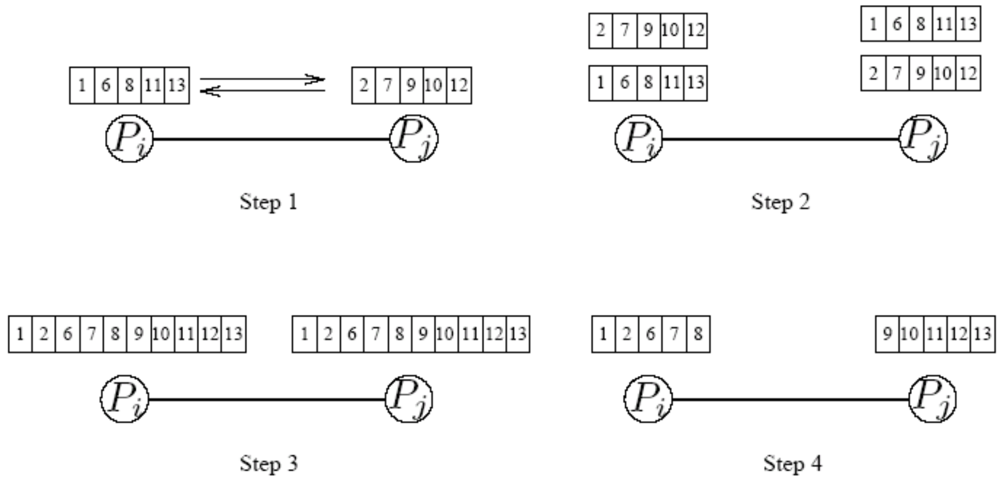


A parallel compare-exchange operation.
Processes P_i and P_j send their elements to each other.
Process P_i keeps $\min\{a_i, a_j\}$ and P_j keeps $\max\{a_i, a_j\}$.

Sorting: Basics

- If each processor has one element, the *compare-exchange* operation stores the smaller element at the processor with smaller *id*.
- 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 $i < j$.

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.

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

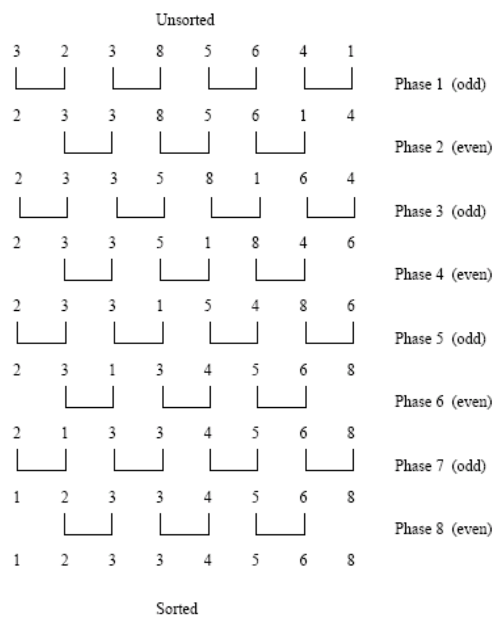
- The complexity of bubble sort is $O(n^2)$.
- Bubble sort is difficult to parallelize since the algorithm has no concurrency.
- A simple variant, though, uncovers the concurrency.

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.

Odd-Even Transposition



Sorting $n = 8$ elements,
using the odd-even transposition sort algorithm.

Odd-Even Transposition

- After n phases of odd-even exchanges, the sequence is sorted.
- Each phase of the algorithm (either odd or even) requires $O(n)$ comparisons.
- Serial complexity is $O(n^2)$.

Parallel Odd-Even Transposition

- Consider the one item per processor case.
- There are *n* iterations, in each iteration, each processor does one compare-exchange (CE).
- The parallel run time of this formulation in the BSP model is $O(n) = O(p)$:
  - $T_{CE} = 1 + l + 1.g + l$
  - $T_p^{(1)} = nT_{CE} = n(1 + 1.g + 2l)$

Parallel Odd-Even Transposition

```
1.   procedure ODD-EVEN_PAR($n$)
2.   begin
3.       $id$ := process's label
4.       for $i$ := 1 to $n$ do
5.       begin
6.           if $i$ is odd then
7.               if $id$ is odd then
8.                   compare-exchange_min($id + 1$);
9.               else
10.                  compare-exchange_max($id - 1$);
11.          if $i$ is even then
12.              if $id$ is even then
13.                  compare-exchange_min($id + 1$);
14.              else
15.                  compare-exchange_max($id - 1$);
16.      end for
17.  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, the compare-exchange operation is replaced by the compare-split operation.

- What is the parallel run time ?
- Derive an expression for the parallel efficiency *E* that clearly shows how *E* depends on *p* and *n*.
- How should the problem size *n* grow with the number of processors *p* to have iso-efficient behaviour?

Parallel Odd-Even Transposition

Parallel run time: (assume *n* is multiple of *p*)

- $T_{CS} = n/p + l + (n/p)g + l = (n/p)(1 + g) + 2l$

  CS = Compare-Split;

- $T_{local\_sort} = c_s\, n/p \log(n/p))$
- $T_p^{(n/p)} = T_{local\_sort} + p\, T_{CS} = c_s\, n/p \log(n/p) + n\,(1+g) + 2pl$
- $E = S/p = T_s / (p\, T_p^{(n/p)})$
- To show how *E* depends on *p* and *n* we can write *E* as follows

$$E = \frac{c_s n \log n}{c_s n \log n - c_s n \log p + pn(1+g) + 2p^2 l}$$

$$E = \frac{1}{1 - O(\frac{\log p}{\log n}) + O(\frac{p}{\log n}) + O(\frac{p^2}{n \log n})}$$

Iso-efficiency & scalability

$$E = \frac{1}{1 - O(\frac{\log p}{\log n}) + O(\frac{p}{\log n}) + O(\frac{p^2}{n \log n})} = \frac{1}{1 + T_{overhead}}$$

- Which term is dominant in $T_{overhead}$ depends on *n, p, g, l*
- Assume that for $n = n_1$ and $p = p_1$ the parallel code runs with a parallel efficiency *E.* Suppose you want to use p_2 processors and you want to achieve the same efficiency.
- If term $O(\frac{p}{\log n})$ is dominant:
  Then the size of the array to be sorted n_2 must satisfy

$$\frac{p_2}{\log n_2} = \frac{p_1}{\log n_1} \Rightarrow \log n_2 = \frac{p_2}{p_1} \log n_1 \Rightarrow n_2 = n_1^{\frac{p_2}{p_1}}$$

N fast growing function of p !

- If term $O(\frac{p^2}{n \log n})$ is dominant:
  Then n_2 must satisfy $n_2 \log n_2 = (\frac{p_2}{p_1})^2 n_1 \log n_1$

17

Iso-efficiency & scalability

$$E = \frac{1}{1 - O(\frac{\log p}{\log n}) + O(\frac{p}{\log n}) + O(\frac{p^2}{n \log n})} = \frac{1}{1 + T_{overhead}}$$

- Analogously:
- Assume that for $n = n_1$ and $p = p_1$ the parallel code runs with a parallel efficiency *E.* Suppose you want to sort an array of size n_2 and you want to achieve the same efficiency.
- If term $O(\frac{p}{\log n})$ is dominant:
  Then p_2 must satisfy $\frac{p_2}{\log n_2} = \frac{p_1}{\log n_1} \Rightarrow p_2 = \frac{\log n_2}{\log n_1} p_1$

p slowly growing function of *n* => more work per proc & larger parallel execution time !

- If term $O(\frac{p^2}{n \log n})$ is dominant:
  Then n_2 must satisfy $p_2 = \sqrt{\frac{n_2 \log n_2}{n_1 \log n_1}} p_1$

18

Quicksort

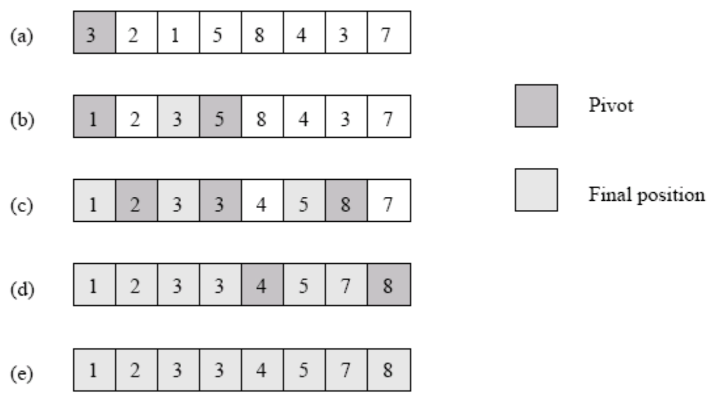
- Quicksort is one of the most common sorting algorithms for sequential computers because of its simplicity, low overhead, and optimal average complexity.
- Quicksort selects one of the entries in the sequence to be the pivot and divides the sequence into two.
  one with all elements less than the pivot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

Quicksort

```
1.    procedure QUICKSORT ($A, q, r$)
2.    begin
3.        if $q < r$ then
4.        begin
5.            $x := A[q]$;
6.            $s := q$;
7.            for $i := q + 1$ to $r$ do
8.                if $A[i] \leq x$ then
9.                begin
10.                   $s := s + 1$;
11.                   swap($A[s], A[i]$);
12.               end if
13.           swap($A[q], A[s]$);
14.           QUICKSORT ($A, q, s$);
15.           QUICKSORT ($A, s + 1, r$);
16.       end if
17.   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 of n elements in 2 (sub)lists of $n/2$ elements.

- In this case, the complexity of quicksort is $O(n\log n)$.

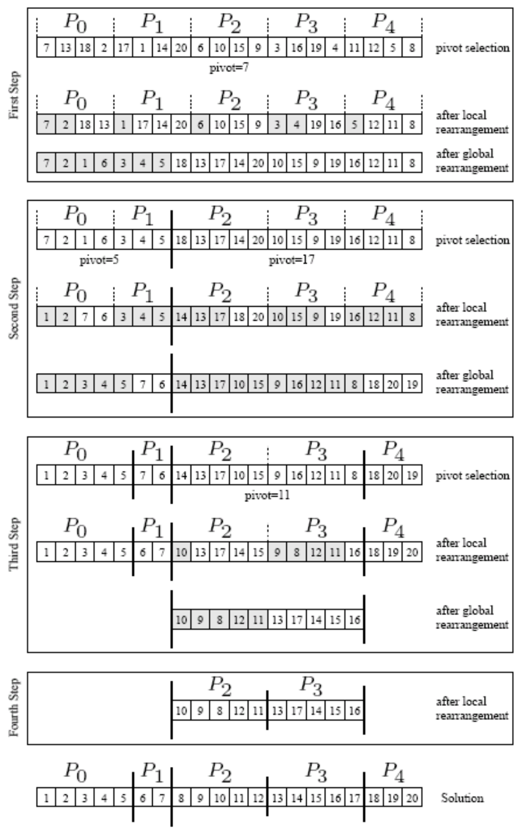
Parallelizing Quicksort

- 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*) !
- 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.

Parallelizing Quicksort: Shared Address Space Formulation

- Consider 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.

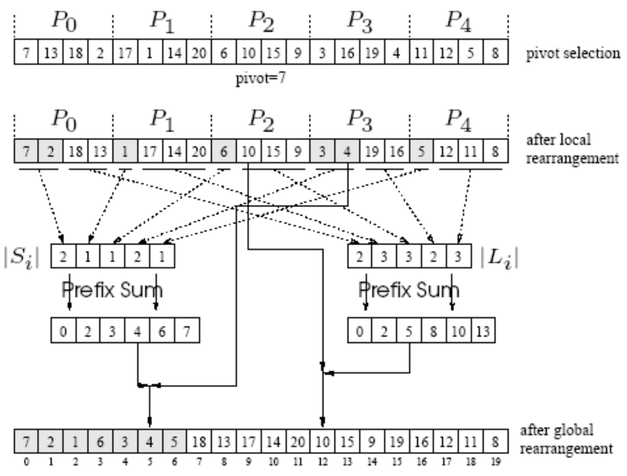
Shared Address Space Formulation



Parallelizing Quicksort: Shared Address Space Formulation

- The only thing we have not described is the 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 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.

Parallelizing Quicksort: Shared Address Space Formulation



Efficient global rearrangement of the array.

Parallelizing Quicksort

- 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.
- The algorithm executes in 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 O(log p) or O(p) (depending on how the broadcast is implemented) , the second O(n/p) ,
the third O(log p) (see next slide), and the fourth O(n/p).
- The overall complexity of splitting an n-element array is
O(n/p) + O(log p).
- The process recurses until there are *p* lists, at which point, the lists are sorted locally.

Parallelizing Quicksort

- Broadcast: simple method: BSP-cost = *$(p-1)g + l$*
  broadcast via recursion or tree: BSP-cost = *$(g + l) \log p$*
- Determine the locations in the globally rearranged array :

  can busing ‘prefix sum’ or ‘partial sums’

  assume array $a = (a_1, a_2, a_3, \ldots, a_n)$

  partial sums s_i ($i = 0,\ldots,n$) $\quad s_i = \sum_{j=0}^{n-1} a_j \quad or \quad s_i = \sum_{j=1}^{n} a_j$

  can be obtained by an algorithm that is a variant of the all-to-all communication

  BSP-cost = O(p) or, when implemented via (concurrent) trees, BSP-cost = *$O(\log p)$*

Parallelizing Quicksort: Message Passing Formulation

- 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.
- It is easy to see that after this step, all elements less than the pivot are in the low half of the machine and all elements greater than the pivot are in the high half.

Parallelizing Quicksort: Message Passing Formulation

- The above process is recursed until each processor has its own local list, which is sorted locally.
- The execution times are: O(log *p*) or O(p-1) for broadcasting the pivot element, O(*n/p*) for splitting the locally assigned portion of the array, O(*n/p*) for exchange and local reorganization.
- It is important to remember that the reorganization of elements is a bandwidth sensitive operation.

Pivot selection

- Pivot selection is more important in the parallel algorithm than in the sequential quicksort algorithm ! Why?
- For random data sets, 'sampling' a subset of the data and determining the optimal pivot (median) of the subset results in good pivot selection.
  Of course this creates an extra overhead.