Parallel Sorting Algorithms

- **Intro: Parallelizing standard sorting algorithms**
- **Bubble sort, odd-even sort**
- **Quicksort**
- **Odd-even Merge-sort**
- **Sorting Networks and bitonic sort**

The problem and some background

➤ Sorting = to order data in some order [e.g. increasingly]

Problem: Given the array $A = [a_1, \ldots, a_n]$ sort its entries in increasing order.

Example

A= [3, 6, 4, 1, 3, 9] ➤ B := [1, 3, 3, 4, 6, 9]

➤ The keys $a_1, .., a_k$ can be integers, real numbers, etc..

➤ There are sequential algorithms which cost $O(n^2)$, $O(n \log n)$, or even $O(n)$. They all have different characteristics and advantages and disadvantages.

➤ Sorting is important and is part of many applications

➤ Results known in sequential case: cannot sort n numbers in less than $O(n \log n)$ comparisons - by comparing keys.

➤ Mergesort has lowest theoretical complexity

➤ Quicksort - is best in practice [even though it can potentially lead to $O(n^2)$ time]

For illustrations, we assume we have only positive integers to sort [no loss of generality]

Assumptions on data

➤ Need to make assumptions on where the data is initially and where it should end at the end. Most common:

Before sorting Data is distributed on the nodes –

After sorting Sorted sub-list should be on each node

Sub-lists are globally ordered in some specific way. [recall Lab2.] –

What speed- can we expect?

➤ Recall $O(n \log n)$ optimal for any sequential sorting without using specific properties of keys.

➤ Best time we can expect based upon a sequential sorting algorithm using n processors is

$$T_{opt,n} = \frac{O(n \log n)}{n} = O(\log n)$$

➤ Difficult to achieve. Do-able but with a very high pre-factor –

Rank-sort

➤ Brute force approach – uses a non cost-optimal method

➤ For each key a, count the number of keys that are $\leq a$. This gives the position where a goes in the sorted list

```
for (i = 0; i < n; i++) {
  k = 0;
  for (j = 0; j < n; j++){
    if (a[j] < a[i]) k++;
  }
  b[k] = a[i];
}
```

✍D1 What happens if there are duplicates? How to fix this?

✍D2 Best time that can be achieved? Needed resources?

Compare Exchange

➤ Let a and b be two items in a list. Compare a and b. If $a > b$ then swap a and b.

➤ Can be generalized to situation when a are b are sublists A and B [e.g., situation when an array is split into n/p equal parts]

Background: Merging two sorted lists

"Given two sorted list A and B, create a sorted list C which contains the items of A and B"

➤ Termed a Merge operation.


```
Step 0:  <------  A ---------> <----  B ---->
         | 1* 3   5   7   9    10 | 2*  4   6   8 |
      C: [-----------------------------------]
```

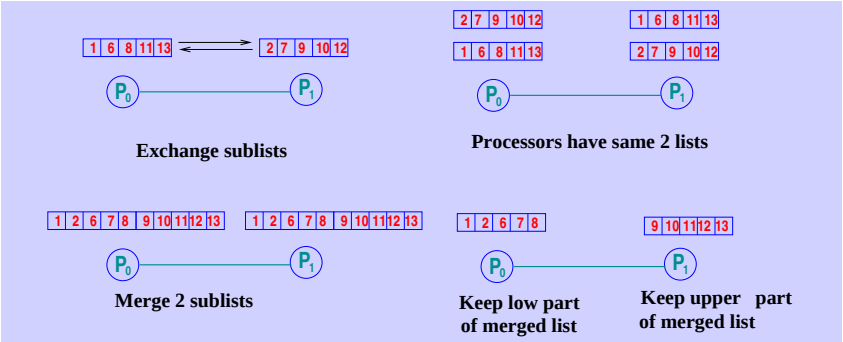
➤ Move Min (A*,B*) to C, then remove the element just moved from its array. Repeat. In the end append left-over to C

```
Step 0:  <------  A ---------> <----  B ---->
         | 1  3  5  7  9   10 | 2   4  6  8 |
      C: [-------------------------------]
Step 1:  <------  A ---------> <----  B ---->
         |    3  5  7  9  10  | 2   4  6  8 |
      C: [ 1-----------------------------]
Step 2:  < -----  A ---------> <----  B ---->
         |    3  5  7  9  10  |     4  6  8 |
      C: [ 1  2--------------------------]
....
Step 9 :  <-----  A ---------> <----  B --->
          |             9  10  |            |
      C: [1   2   3   5   6   7  8   9   10 ]
                                  ^^^^^^
                                append rest of A
```


Compare-Split

Problem: P_0 and P_1 each has 1 sublist. Want lower half of combined list in P_0, the other half in P_1.

Solution: (1) Exchange the sublists. Each PE has combined list. (2) Merge the combined list in each PE; (3) Retain the lower part in P_0 and upper part in P_1.



Bubble Sort: [sequential]

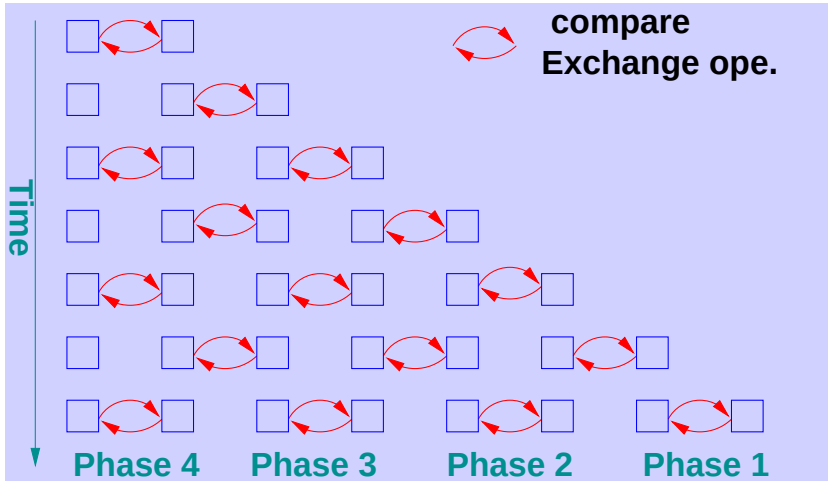
* First, largest number moved to the end of list by a series of compares and exchanges: a_1, a_2, a_2, a_3, ..., a_{n-1}, a_n.
* Repeat with sequence $a_1, \cdots, a_{n-1}$.
* Larger numbers move toward end [like bubbles]

```
for (i = 1; i < n; i++) {
  for (j = 0; j < n-i; j++)
    compare_exch(&a[j],&a[j+1]);
```

➤ $O(n^2)$ comparisons.

Parallel Implementation

Several phases [instances of outer loop] can run in parallel – as long as one phase does not overtake next



Odd-even Sort

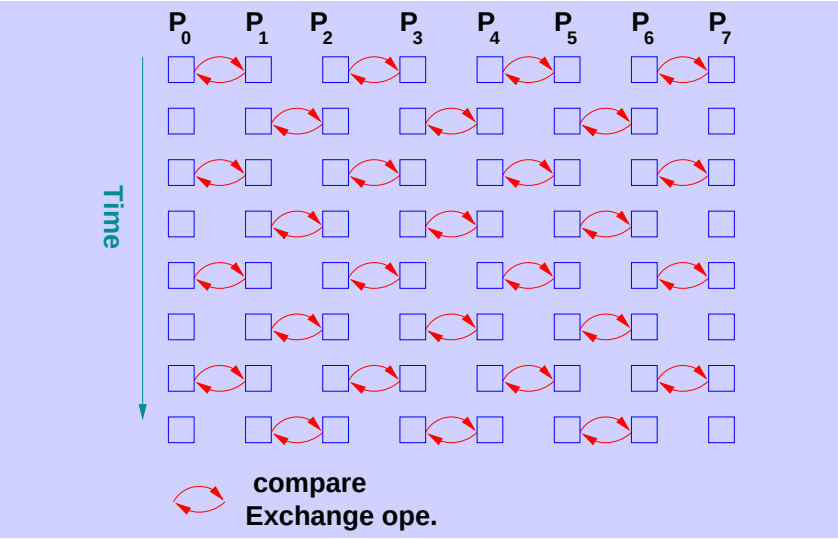
- Variation of bubble sort – Two alternating phases, even phase and odd phase.

Even phase Even-numbered keys (a_{2i}) are compare- exchanged with next odd-numbered keys (a_{2i+1})

Odd phase Odd-numbered keys (a_{2i-1}) are compare - exchanged with next even-numbered keys indices (a_{2i})

- Requires n steps
- For $p = n$, efficiency is $log(n)/n$ [with Quicksort on one PE]
- Can be implemented for $p \leq n$.

Illustration for case $n = p = 8$



Odd-even Sort- general case $n > p$

- Given p sublists of n/p keys in each processor
- Sort each part in each processor –
- Then perform odd-even algorithm in which each compare-exchange (pairs of items) is replaced by a compare-split (pairs of sublists)

Total Cost: Let $m = n/p$

* Initial sort: $O(m \log m)$

* p phases ($p/2$ odd + $p/2$ even) of exchanges followed by merges: $O(p \times m) = O(n)$ for communication + same thing for merge.

* Total $T(n,p) = O(\frac{n}{p} \log \frac{n}{p}) + O(n)$.

* Efficiency

$$E(n,p) = \frac{1}{1 + O\left(\frac{p}{\log n}\right) - O\left(\frac{\log p}{\log n}\right)}$$

- To keep efficiency constant we need n to increase exponentially with p.
- Very poor scalability – works fine when $n \gg p$.

Quick-Sort – Background

- One of the most efficient algorithms in practice
- Worst case performace $= O(n^2)$
- Average case performance $= O(n \log_2)$

Main idea of QuickSort: Divide and conquer

Pivot: Select a pivot element $t = a(k)$

Split: Rearrange keys so that: $a(mid) = t$; $a(i) < t$ if $i < mid$; and $a(i) > t$ if $i > mid$

Recursive call: Call QuickSort on the two subsets $a_1, \ldots, a_{mid-1}$ and $a_{mid+1}, \ldots, a_n$

Main ingredient of QuickSort: Split

- Select an element from the array, e.g., $x = a_1$;
- Start with split-point = 1. Call sp the split-point.
- Scan array from 2 to n, and whenever an element a_j smaller than x is found, add one to sp and swap a_j and a_{sp}.
- In the end swap a_1 and a_{sp}.

Pseudocode for split

```
sp = 1;
x  = a[sp];
for j = sp+1 to n do:
    if (a[j] < x) swap a[++sp] and a[j];
endfor
swap a[1] and a[sp]
```


Example

- Let A = [10 12 9 15 5 17 8] and pivot = 1.

```
-------------------------------------------------
>10< 12* 9   15    5   17   8 --> j=2: sp =1 no sw
>10< 12   9* 15    5   17   8 --> j=3: sp+=1 & swap ->
 10 >9< 12   15*   5   17   8 --> j=4: no change
 10 >9< 12   15   5*   17   8 --> j=5: sp+=1 & swap ->
 10   9   >5< 15   12   17* 8 --> j=6: no change
 10   9   >5< 15   12   17   8*--> j=7: sp+=1 & swap ->
 10   9    5   >8< 12   17 15 --> Swap a(1) and a(sp):
-------------------------------------------------
  8   9    5   >10<   12   17   15   --> DONE.
-------------------------------------------------
```

- Quick-sort processes the lists [8 9 5] and [12 17 15] recursively.

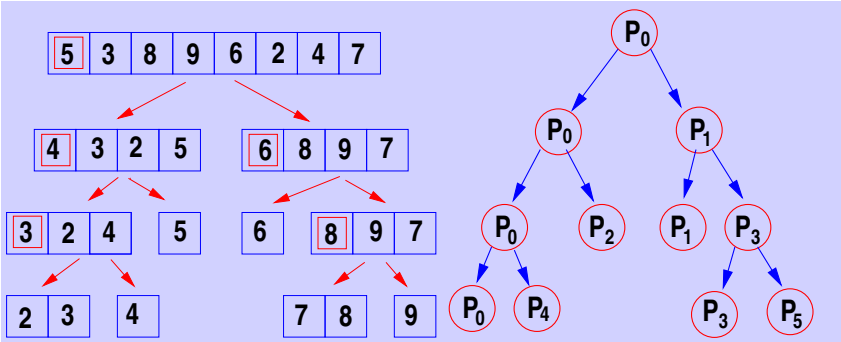
Parallelizing Quick-Sort

- Trivial implementation : Use a tree structure –

Important: Assumes Data is initially on one processor.

Steps:

1. In PE0: Find a splitting key.
2. In PE0: Split array in two parts
3. In PE0: Send one half of the data on to PE1.
4. Repeat above *recursively* for sublists in PE0 and PE1



➤ Major limitation: initial step takes $O(n)$ - it is done on one processor. Total is $O(n)$ - speed-up is at most $\log n$.

➤ This is because data is assumed to be initially in one PE

➤ Practical versions in which list is initially split into p Processors: use the hypercube model [See pages 8-1 to 8-3 of lecture notes # 8 for an illustration]

➤ Recursively sort halves. Message passing version involves explicit exchange of data.

➤ Shared memory versions are easy: (1) do a local split (2) rearrange whole array (3) subdivide the 2 halves into $p/2$ parts and (4) decide which processors will work on each of the subparts.