Efficient (Parallel) Sorting

- One of the most frequent operations performed by computers is organising (sorting) data.
- The access to sorted data is more convenient/faster.
- There is a constant need for good sorting algorithms including sequential, parallel and distributed solutions.
- There is a plethora of sorting algorithms. We already know that one can use heaps for sorting. Here we focus on two sorting procedures including quick-sort and merge-sort.

Merging to ordered sequences

- The key to merge-sort is merging procedure merge, s.t., having two input sequences:
  - \( A = (a_1 \leq a_2 \leq \cdots \leq a_m) \) and \( B = (b_1 \leq b_2 \leq \cdots \leq b_n) \)
  - it produces combined \( C = (c_1 \leq c_2 \leq \cdots \leq c_{m+n}) \)

Example:
\[ A = < 3, 8, 9 > \quad B = < 1, 5, 7 > \]
\[ \text{merge}(A, B) = < 1, 3, 5, 7, 8, 9 > \]

Merging (cont.)

- Pick the minimum:
  - For example, in the first step, pick 1 from A and 5 from B.

Merging (cont.)

- And save it here:
  - In this case, 1.

- Pointers:
  - Move the pointers to the next element:
    - A moves to 10, B moves to 5.

- X: 3 10 23 54
- Y: 5 25 75
- Result: 1 3

- X: 3 10 23 54
- Y: 5 25 75
- Result: 1 3
Merging (cont.)

X: 10 23 54 Y: 5 25 75

Result: 1 3 5

Merging (cont.)

X: 10 23 54 Y: 5 25 75

Result: 1 3 5 10

Merging (cont.)

X: 23 54 Y: 25 75

Result: 1 3 5 10 23

Merging (cont.)

X: 54 Y: 25 75

Result: 1 3 5 10 23 25
Divide-and-Conquer Method

- A very natural recursive approach
  - Divide
    - if the input size is small then solve the problem directly;
    - otherwise divide the input data into two or more disjoint subsets
  - Recur
    - recursively solve the sub-problems associated with the subsets
  - Conquer
    - take the solutions to the sub-problems and merge them into a solution to the original problem

Merge-Sorting

- Divide: if input sequence $S$ has 0 or 1 element then return $S$; otherwise split $S$ into two sequences $S_1$ and $S_2$, each containing about $1/2$ elements of $S$
- Recur: recursively sort sequences $S_1$ and $S_2$
- Conquer: Put the elements back into $S$ by merging the sorted sequences $S_1$ and $S_2$ into a single sorted sequence

Single run of merge procedure produces combined sorted sequence. Thus the time complexity is linear $O(m+n)$. 

Single run of merge procedure produces combined sorted sequence. Thus the time complexity is linear $O(m+n)$. 

Divide-and-Conquer Method

- A very natural recursive approach
  - Divide
    - if the input size is small then solve the problem directly;
    - otherwise divide the input data into two or more disjoint subsets
  - Recur
    - recursively solve the sub-problems associated with the subsets
  - Conquer
    - take the solutions to the sub-problems and merge them into a solution to the original problem

Merge-Sorting

- Divide: if input sequence $S$ has 0 or 1 element then return $S$; otherwise split $S$ into two sequences $S_1$ and $S_2$, each containing about $1/2$ elements of $S$
- Recur: recursively sort sequences $S_1$ and $S_2$
- Conquer: Put the elements back into $S$ by merging the sorted sequences $S_1$ and $S_2$ into a single sorted sequence

Single run of merge procedure produces combined sorted sequence. Thus the time complexity is linear $O(m+n)$.
Merge-Sorting (top down approach)

Divide the input sequence evenly to $S_1$ & $S_2$

Recur

Conquer by merging sorted sequences

Merge-Sorting (example)

Recall that merging two sorted sequences $S_1$ and $S_2$ takes $O(n_1+n_2)$ time, where $n_1$ is the size of $S_1$ and $n_2$ is the size of $S_2$

The depth of the recursion is $O(\log n)$ due to the halving process

Thus merge-sort runs in $O(n \log n)$ time in the worst (and average) case

Merge-Sorting (analysis)

Merge-Sorting (analysis)
Quick-Sort

- Divide if $|S| > 1$, select a pivot value $x$ in $S$ and create three sequences: $L$, $E$ and $G$, s.t.,
  - $L$ stores elements in $S < x$
  - $E$ stores elements in $S = x$
  - $G$ stores elements in $S > x$
- Recur recursively sort sequences $L$ & $G$
- Conquer put sorted elements from $L$, $E$ and finally from $G$ back to $S$.

Quick-Sort Tree

- Divide the sequence $S$ using random pivot $x$
- Recur
- Conquer by concatenating sorted sequences $L(<x)$ and $H(>x)$

Quick-Sort (example)

Quick-Sort (worst case)

- Let $s_i$ be the sum of the input sizes of the nodes at depth $i$ in a quick sort tree $T$
- $s_i \leq n - i$ (and $s_i = n - i$ when use of pivots lead always to only one nonempty sequence: either $L$ or $G$)
- The worst-case complexity is bounded by $O(n^2)$.
  \[ O \left( \sum_{i=0}^{n-1} s_i \right), \text{ which is } O \left( \sum_{i=0}^{n-1} (n - i) \right) \text{ that is, } O \left( \sum_{i=1}^{n} i \right) \]
Quick-Sort (randomised algorithm)

- **Thm:** the expected running time of randomised (pivot is chosen in random) quick-sort is $O(n \log n)$
- **Proof:**
  - The expected number of times that a fair coin must be flipped until it shows heads $k$ times is $2^k$.
  - Randomly chosen pivot is right if neither of the groups $L$ nor $G$ is $\geq \frac{3}{4} |S|$
  - The probability of a success in choosing a right pivot is $\frac{1}{2}$
  - A path in quick-sort tree can contain at most $\log_{4/3} n$ nodes with right pivots
  - Hence, the expected length of each path is $2\log_{4/3} n$

Lower Bound (comparison-based model)

- In comparison-based model the input elements can be compared only with themselves and the result of each comparison $x_i \leq x_j$ is always yes or no
- **Thm:** the running time of any comparison-based sorting algorithm is $\Omega(n \log n)$ in the worst case
- **Proof:**
  - Sorting of $n$ elements can be identified with recognising a particular permutation of $n$ elements
  - There are $n! = n \cdot (n-1) \cdot \ldots \cdot 2 \cdot 1$ permutations of $n$ elements
  - Each comparison splits a group of permutations into two groups (one that satisfies the inequality and one that doesn’t)
  - In order to ensure that the size of each group of permutations is brought down to one we need $\log_2(n!) > \log (n/2)^{n/2} = \log n^2 = \Omega(n \log n)$ comparisons
List ranking and prefix sums

- In the link ranking problem one is expected to compute for each element its distance to the front of the list.
- In the prefix sum problem one is expected to compute for each prefix of the list the sum of the keys stored in this prefix.
- Computing prefix sums with all keys of value 1 is equivalent to the link ranking problem.

All read at distance $2^0$ & add to their own values

All read at distance $2^1$ & add to their own values
List ranking and prefix sums can be computed in $O(\log n)$ time when $n$ is the size of the input:

- During every single round we increase knowledge about preceding block of $2^i$ positions in $O(1)$ time.
- After $O(\log n)$ rounds of doubling the job is done.

We need also another tool that will allow us to collect and distribute information to all processors also in $O(\log n)$ time.
Information dissemination

- $P_0$ informs neighbour at distance $2^0$

- $P_0, P_1$ inform neighbours at distance $2^1$

- $P_0, P_1, P_2, P_3$ inform neighbours at distance $2^2$

- $P_0, ..., P_6, P_7$ inform neighbours at distance $2^3$
Information collection/dissemination

- The process of collection of information is done by reversing communication (direction of arrows) used during information dissemination.
- Both processes take time $O(\log n)$.
- This means that processors can all agree on simple decisions (via exchanging small messages), e.g., “is there any work left to do?” in time $O(\log n)$.

Parallel Quick-Sort

- The complexity analysis of parallel quick-sort:
  - Every stage takes at most time $O(\log n)$
  - Expected number of stages is $O(\log n)$
  - The total computation time is $O(\log^2 n)$
  - The number of processors needed is $n$
  - The total work is $O(n \log^2 n)$
- One can reduce work to optimal using $n/\log n$ processors.

Parallel Quick-Sort

- Sequence $S[1\ldots r]$ is being sorted
  - The local size of the input $n = r-l+1$
- Each $P_i$ ($i=L\ldots r$) picks value $S[i]$ with prob. $1/n$
  - A unique pivot value $p$ is communicated to all (if none or more values are picked the process is repeated)
- The values from $S[i\ldots r]$ are distributed to $L[i\ldots r]$ & $H[i\ldots r]$
- Using list ranking and prefix sums compute the ranks of values in $L$ and $H$
- The number of values #L in L is communicated
- The values are copied back to $S$ as follows
  - Value with rank $n$ in $L$ is moved to $S[i+\ldots n]$
  - The pivot $p$ is moved to $S[i+\#L]$
  - Value with rank $i$ in $H$ is moved to $S[i+\#L+\ldots j-1]$
- Sort recursively $S[1\ldots i-\#L+1]$ & $S[i+\#L+1\ldots r]$

Parallel Merge-Sort

- Assume two halves $L$ & $H$ of $S[1\ldots r]$ are already (recursively) sorted
  - The local size of the input $n = r-l+1$
- Using binary search compute a rank of each $L$ value in the other half $H$, and vice versa
- Combine (add) the two ranks (from $L$ and $H$) to find the new position in the sorted sequence.
Parallel Merge-Sort

- The complexity analysis of parallel merge-sort
  - Each stage (binary search) takes at most time $O(\log n)$
  - The number of recursive stages is $O(\log n)$
  - The total computation time is $O(\log^2 n)$
  - The number of processors needed is $n$
  - The total work is $O(n\log^2 n)$
  - One can reduce work to optimal using $n/\log n$ processors