CSE4502/5717: Big Data Analytics
Prof. Sanguthevar Rajasekaran
Lecture 7 – 02/14/2018; Notes by Aravind Sugumar Rajan

Recap from last class:

Prim's Algorithm:

- Grow a subtree by adding one edge to the subtree at any time
- We use NEAR Data Structure to achieve this

Analysis:

Step 1: Picking the lightest edge in E takes $\frac{|E|}{B}$ I/O operations;

Step 2: Calculating NEAR values of all the vertices takes $\frac{|V|}{B}$ I/O operations;

Step 3: Assuming that the size of the internal memory = $\theta(|V|)$, the priority queue can be kept in memory; so, no I/O operations are involved in Step 3.

Step 4: Let d_u be the degree of the node u, u ∈ V - {a, b}. In Step 4 we have to access the neighbors list for every node (other than a and b) in V. As a result, Step 4 takes no more than $\sum_{u\epsilon V} \left\lceil \frac{d_u}{B} \right\rceil$ I/O operations.

$$\sum_{u\epsilon V} \left\lceil \frac{d_u}{B} \right\rceil \leq 2\frac{|E|}{B} + |V|$$

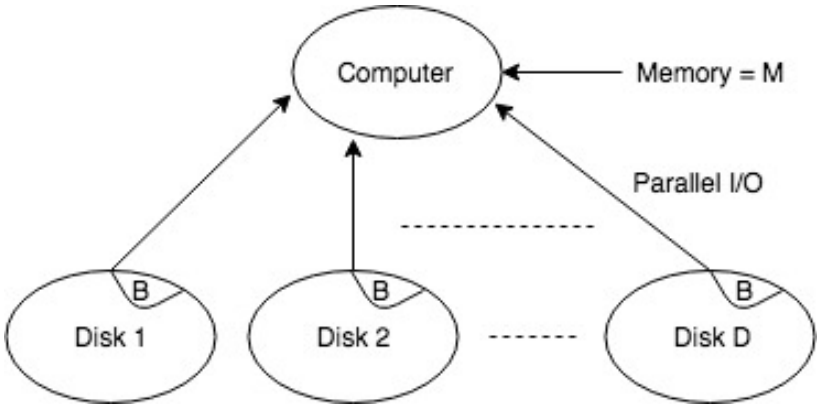
$\therefore Total\ number\ of\ I/O$ operations = $O\left(\frac{|E|}{B} + |V|\right)$

This is asymptotically optimal for dense graphs, e.g., when $|E| \geq B|V|$.

Best known algorithms:

- A Functional Approach to External Graph Algorithms, Abello et al., 2002
  - A Randomized algorithm that takes $\hat{O}\left(\frac{S(|V|+|E|)}{B}\right)$ I/O operations, where S(k) stands for the number of I/O operations needed to sort k elements.
- On external-memory MST, SSSP and multi-way planar graph separation, Arge et al.,2004
  - A deterministic algorithm that takes $O\left(\frac{S(|V|+|E|)}{B} \log\log\frac{|V|}{M}\right)$ I/O operations.

Parallel Disks Model (PDM): This model has been proposed to deal with I/O bottlenecks that could arise while dealing with massive datasets. We assume that there are D disks (D being more than 1) and a (sequential or parallel) computer with a core memory of size M. The goal is to devise algorithms for solving various problems on the PDM such that the number of parallel I/O operations is minimized.



In one parallel I/O operation, we can bring a block of data from each of the D disks. We'll see how to sort elements on a PDM.

Why is sorting important?
Estimates indicate that there is approximately a 40% probability that any machine is sorting at any particular time.

Problem: Sort n elements
Input: Each disk has $\frac{n}{D}$ elements;
Output: Sorted sequence striped across all the disks;

The difficulty in this model is to ensure that in every parallel I/O operation we bring one (useful) block from every disk.

Many known PDM sorting algorithms are based on k-way merge, for some suitable value of k (no more than D).

<u>Assumptions:</u>

- M = $\theta(BD)$

<u>A typical PDM sorting algorithm:</u>

Step 1: Form runs of length M each; this takes 1 pass through the data; we have $\frac{n}{M}$ runs;
Step 2: Use k-way merge to merge these $\frac{n}{M}$ runs; It is desirable to keep k as close to D as possible.

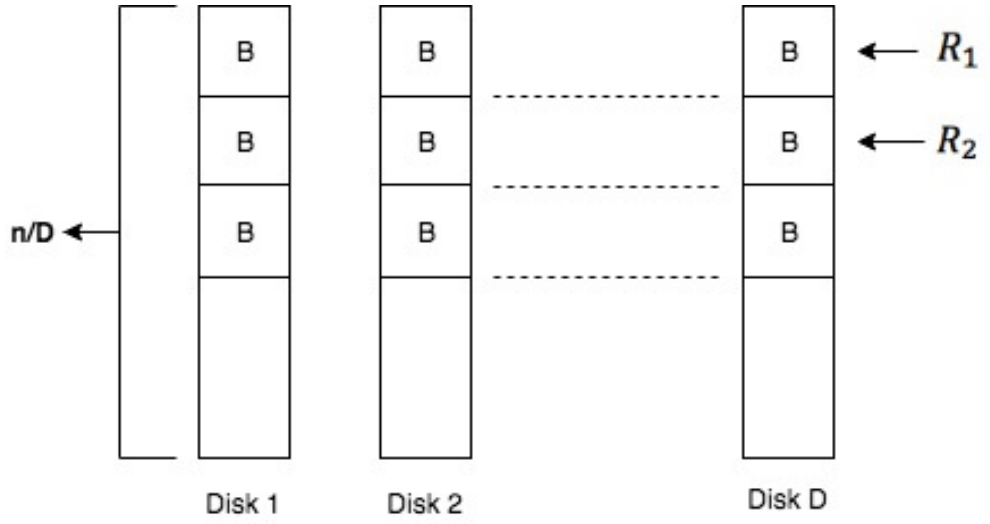
A known lower bound for sorting:

$\Omega\left(\frac{n}{DB}\frac{\log\frac{n}{M}}{\log\frac{M}{B}}\right)$ parallel I/O operations are needed to sort n elements on the Parallel Disks Model.

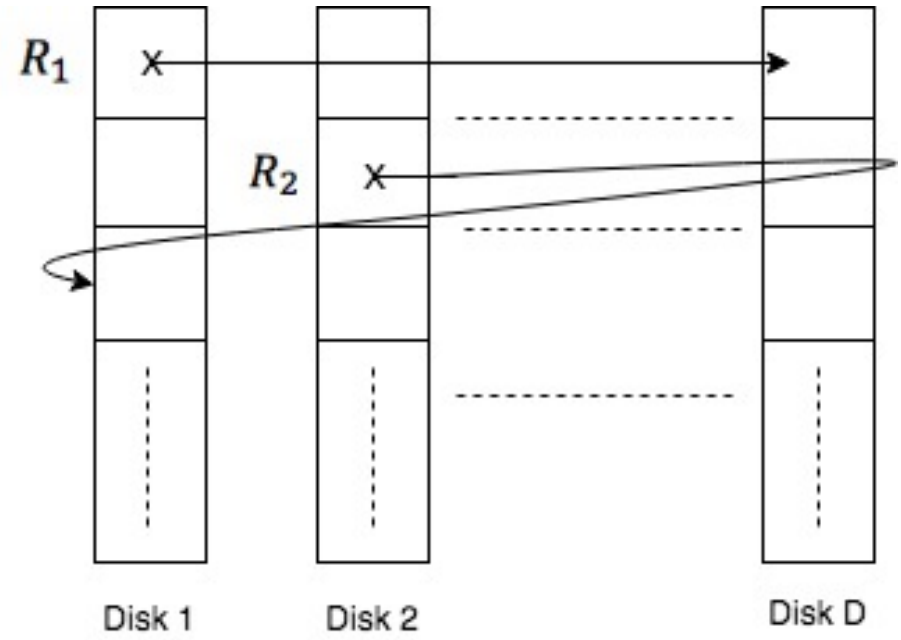
Note: One pass through the data means $\frac{n}{DB}$ I/O operations.

Striping the data: Each run is stored in the disks such that the first block is stored in one disk, the next block is stored in the next disk, and so on. This process is called 'striping the data'.

Here is one way of striping:



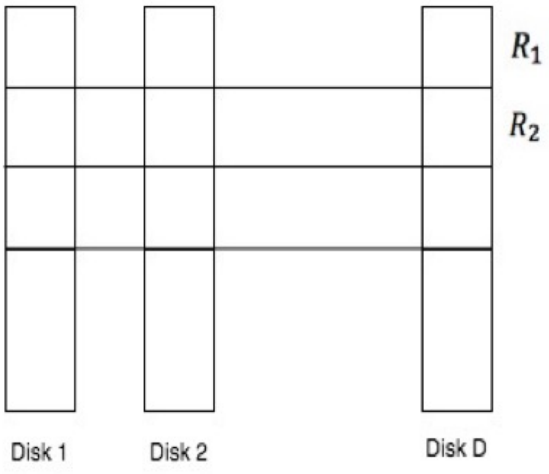
Consider the case where we have D runs. If we want to get the leading block from each of the runs and if we use the above way of striping, then we will need D I/O operations, since all the leading blocks are in the first disk! We can avoid this inefficiency, by using the following approach:



Disk Striped Mergesort (DSM): is a simple algorithm that stripes all the runs starting from the first disk.

-It uses R-way merge, where R = $\frac{M}{DB}$ (a constant).

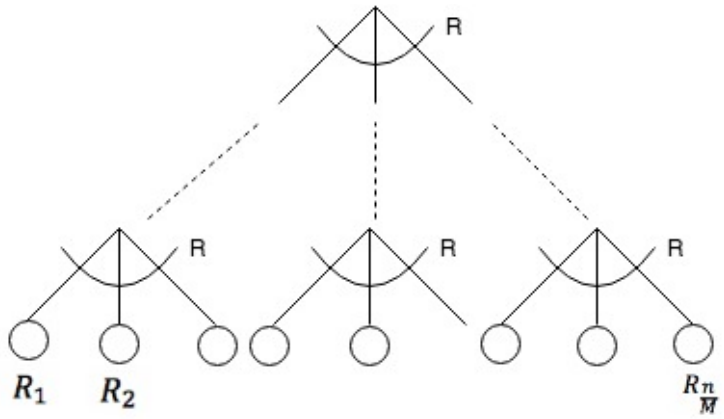
Start by forming runs of length of M each.
Bring BD elements from each run.



Start merging them.
When BD elements are ready in the output buffer, write them onto the disks;
When we run out of elements from any run, bring BD elements from that run;

The number of I/O operations taken by the algorithm = $O\left(\frac{n}{DB}\frac{\log(\frac{n}{M})}{\log(\frac{M}{DB})}\right)$.

R-way merge:



No of parallel I/O operations = $\frac{n}{DB}\frac{\log\left(\frac{n}{M}\right)}{\log R}$.
Since R = $\theta(1)$ for DSM, this algorithm is not asymptotically optimal.

Example:

n = M^c for some constant C

Lower bound for sorting = $\Omega\left(\frac{n}{DB}\frac{\log M}{\log\frac{M}{B}}\right)$ I/O operations.
No. of I/O operations for DSM = $\Omega\left(\frac{n}{DB}\log M\right)$, which is too much.

(l, m)-mergesort is a simple algorithm. We introduce the odd-even merge sort and the s^2-way merge sort before discussing the (l, m)-merge sort.

Odd-Even Mergesort:

Let X = $k_1, k_2, \ldots\ldots, k_n$

Algorithm:

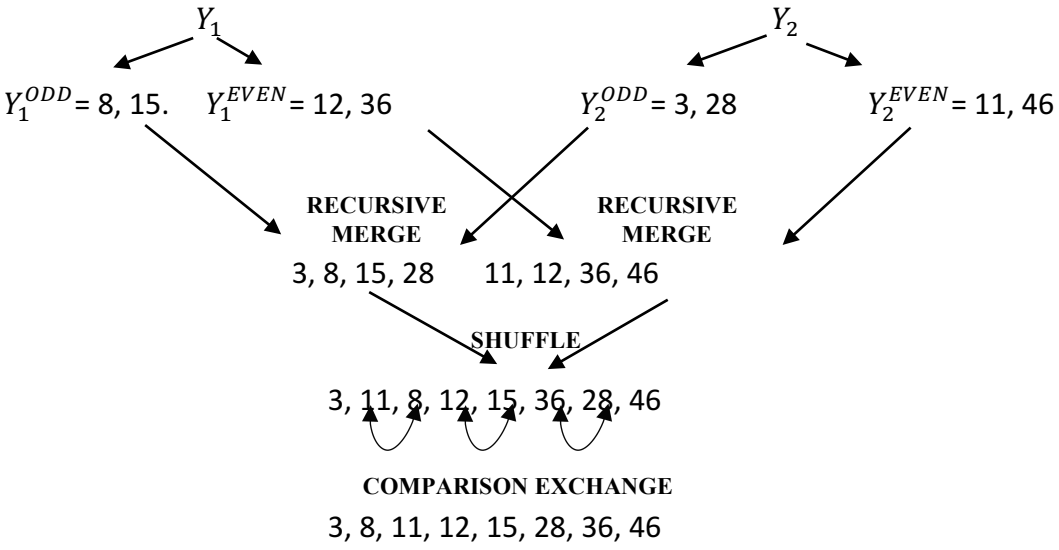
Partition X into X_1 & X_2 with $|X_1| = |X_2| = \frac{n}{2}$;
Sort X_1 & X_2 recursively to get Y_1 & Y_2, respectively.
Merge Y_1 & Y_2 using odd-even merge.

Odd-Even Merge:

Input: Two sorted sequences
$Y_1 = a_1, a_2, \ldots\ldots, a_n$ &
$Y_2 = b_1, b_2, \ldots\ldots, b_n$
Output: Merge of Y_1 , Y_2

Example:

Y_1 = 8, 12, 15, 36 Y_2 = 3, 11, 28, 46



Algorithm:

Let $Y_1 = a_1, a_2, \ldots, a_n$ & $Y_2 = b_1, b_2, \ldots, b_n$;

Step 1: Let $Y_1^{ODD}=a_1, a_3, \ldots, a_{n-1}$
$Y_1^{EVEN}=a_2, a_4, \ldots, a_n$
$Y_2^{ODD}=b_1, b_3, \ldots, b_{n-1}$
$Y_2^{EVEN}=b_2, b_4, \ldots, b_n$
} Unshuffle operation

Step 2: Recursively merge Y_1^{ODD} and Y_2^{ODD} to get $Z_1 = c_1, c_2, \ldots, c_n$;
Recursively merge Y_1^{EVEN} and Y_2^{EVEN} to get $Z_2 = d_1, d_2, \ldots, d_n$;
Step 3: Shuffling operation: Shuffle Z_1 and Z_2 to get Q = $c_1, d_1, c_2, d_2, \ldots, c_n, d_n$;
Step 4: Perform one compare exchange operation as shown below:

$c_1, d_1, c_2, d_2, \ldots\ldots, c_{n-1}, d_{n-1}, c_n, d_n$

Zero-One Lemma:
If an oblivious comparison-based sorting algorithm correctly sorts every sequence of length n that has only zeros and ones, then, it correctly sorts every sequence of length n of arbitrary elements.

Oblivious comparison: Comparison based on position of keys only;
An example for a sorting algorithm that is not oblivious will be bucket sort. We'll use zero-one lemma to prove the correctness of the odd-even merge sort algorithm.