Yaira K. Rivera Sánchez **CSE-5095-005** **January 28, 2014**

CLASS NOTES
Research Topics in Big Data Analytics
Prof. Sanguthevar Rajasekaran

Recap of the last class:

- Two categories of models of parallel computing were introduced, namely, the Fixed Connection Machines and the Shared Memory or Parallel Random Access Machines (PRAMs). Different kinds of PRAMs were defined. The slow-down lemma was stated. We saw how to compute the maximum of *n* given numbers in parallel. Finally, we discussed a CREW PRAM algorithm for the prefix computation problem.

Out-of-Core Computing:

- When we have too much data we may not be able to store all of them in the main (i.e., core) memory of the computer that we are using. Most of the data may have to be stored in secondary storage devices such as disks. We can only bring to the core memory a portion of the data at any given time, do some processing on it and store the (partial results) in the disk.

- There is a hierarchy in memory devices.

- A hierarchy of memory devices could be: Registers, Instruction Cache, L1 Cache, L2 Cache, Core Memory, Disk.

- The devices of the list above are in increasing order in terms of the time they take to access data. Therefore, the first device is the fastest one to access data and the last device shown on the list is the slowest one to access data.

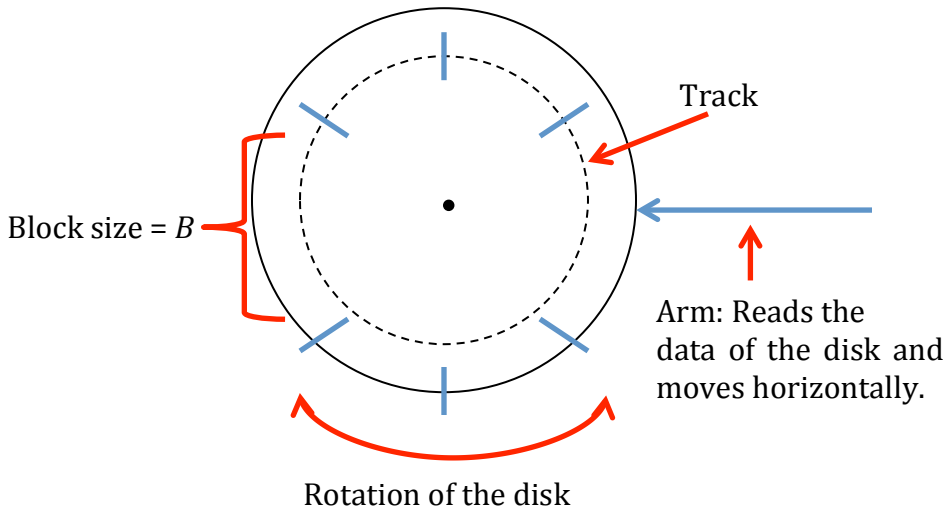
- **The time it takes to access data in the following devices are:**
  - Registers –> Nanosecond (10^{-9})
  - Core Memory -> Several Nanoseconds
  - Disk -> Several Milliseconds
  - SSDs -> Microseconds

- It pays to minimize the number of I/O operations since accessing data in the disk takes more time than having the data in a higher place on the list above.

- We can let the OS handle the I/O operations on the disk(s). However, this may be highly inefficient. The OS typically uses prefetching or caching to minimize I/O operations. We can get vastly better performances by explicitly handling I/O operations in our algorithms.

- **Definition:** An Out-of-Core algorithm is one where the algorithm explicitly dictates how to handle the I/O's. It focuses on trying to minimize the number of I/O's.

- **Representation of a disk:**



- I/O operations happen in units of *blocks.* Due to the latency involved in seeking the right track, moving the arm, etc., it helps to move more than one records in any single I/O operations. This is why a block is involved in any I/O. A block consists of many records. We let B denote the size of a block.

Sorting:

- **INPUT**: $X = k_1, k_2, \dots, k_n$
- **OUTPUT**: Sorted order of X.

- **Fact:** Sorting n keys in core requires $\Omega(n \log n)$ comparisons.

- **Lemma:** Sorting n keys (residing in a disk) needs $\Omega\left(\frac{n}{B}\frac{\log(n/M)}{\log(M/B)}\right)$ I/O operations.

- Here M is the core memory size and B is the block size.

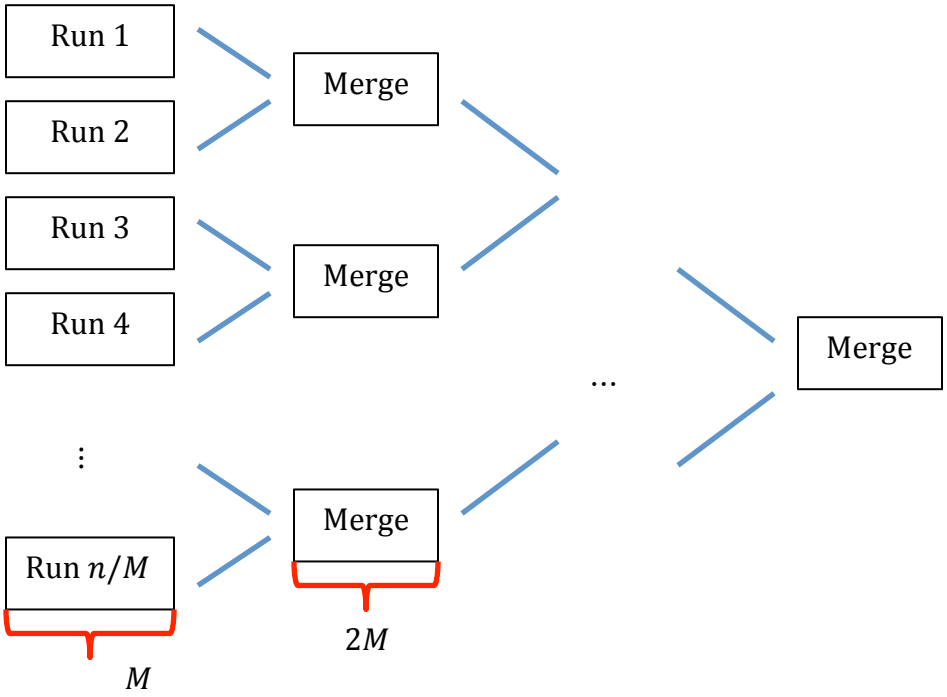
- **Definition:** One pass through the data refers to n/B I/O operations. Each input key is brought into core memory exactly once (in one pass).

Sorting Algorithm:

- **First attempt:**

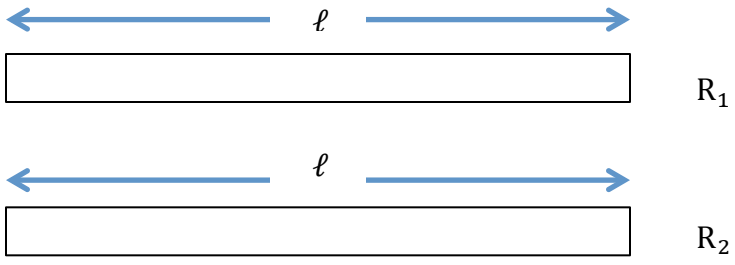
1) Do one pass through the data and form runs of length M each. A run is a sorted subsequence.
- We have to merge these n/M runs.

2) We can use 2-way merge:



- $2M$ is the length of the run obtained by merging two runs of length M each.
- At each level of merging, the number of elements in any run will keep increasing by a factor of 2 until we are left with two runs of length $n/2$ each. When we merge these two we get sorted X.

- Consider the merging of two runs of length ℓ each:



- We keep *M/2B* blocks of R_1 and *M/2B* blocks of R_2 in the main memory. Merge these in the computer.

- When *B* keys in the output are produced, output this block in the disk.

- When *B* keys have been consumed from any run, we do one read (i.e., one block) of that run from the disk .

- **Example 1:**

  R_1: 5, 11, 15, 17, 28, 32, 45

  R_2: 3, 8, 21, 23, 35, 42, 75, 76

  - Let *M*=8 and *B*=2. To begin with the core memory has 5,11,15,17 from R_1 and 3,8,21,23 from R_2. We start merging these two. The first two output elements are 3 and 5. These are output to the disk as a part of the merged run. The next element output is 8. At this point, we have used a block of R_2 and hence the next block of R_2 (i.e., 35,42) will be read into the core memory. The next element output is 11. We write 8,11 into the disk. We also read the next block of R_1; and so on.

- **Example 2:**

  R_1: 1, 2, 3, 4, 5, 6, 7, 8

  R_2: 9, 10, 11, 12, 13, 14, 15, 16

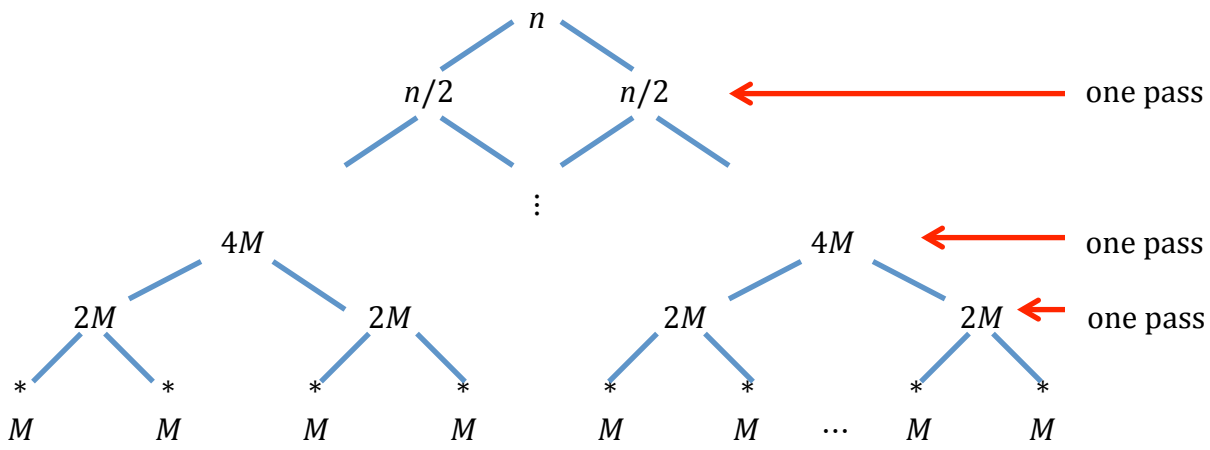
  - **To begin with:**
    - Core memory:

1	2	3	4	5	6	7	8				
9	10	11	12					13	14	15	16

    To begin with we have 4 elements from each run in the core memory. The first two elements output are 1 and 2. They are written to the disk. We then read 5 and 6 from R_1; and so on.

- **Analysis:**

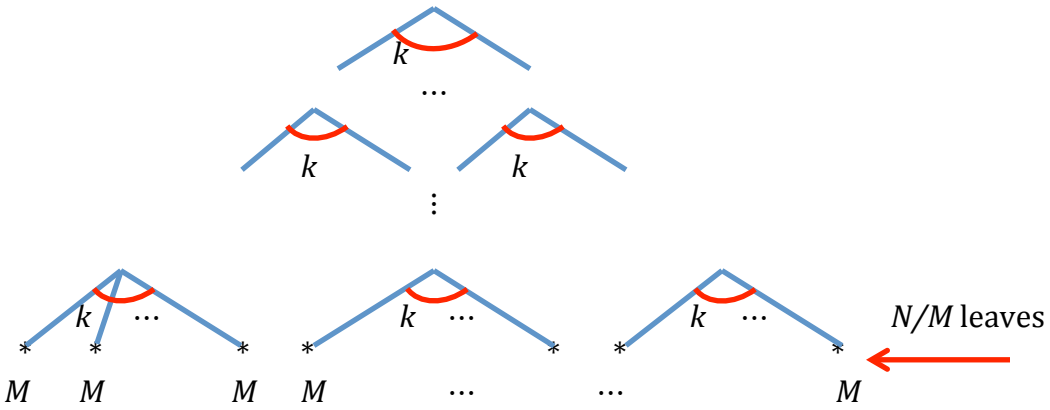
- To merge two sequences of length l each, the number of read I/O operations is equal to $2l/B$ (i.e. we only do one pass through these two runs).



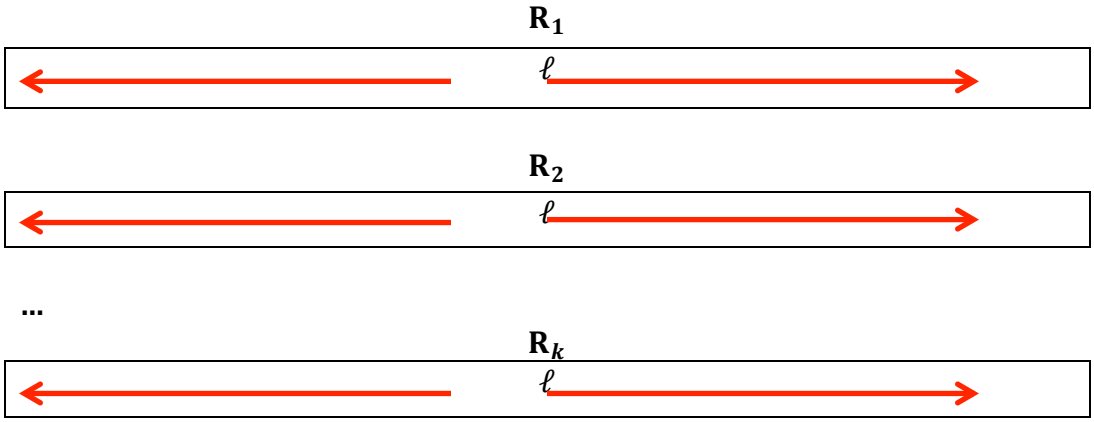
⇨ **The total number of passes is:** $\log(n/M) + 1$

⇨ **The number of I/O's is:** $\frac{n}{B}\left(\log\left(\frac{n}{M}\right) + 1\right)$.

- **Algorithm #2:**
  - A natural extension of the previous algorithm is to merge k runs at a time (for some k>2).
  - We have a k-ary tree:



  - Consider any node in the tree. We merge k runs of length ℓ each:
  -



1) Bring $M/(kB)$ blocks from each run into the core memory to begin with.

2) Start merging them. When a block of output is ready, write it into the disk. When a block from any run has been consumed, read one block from that run.

3) Repeat step 2 as needed.

- **Analysis:**
  - The height of the tree is equal to $\log_k(n/M)$, which is equal to $\frac{\log(n/M)}{\log k}$.
  - We can choose $k = M/B$.

    In this case, each level of mergings in the tree can be done in one pass through the data.
  - When k is greater than the previous value, we can only keep a fraction of each block in core memory at any given time.
  - The larger the value of k, the better it is for us. Nevertheless, there is a limit in how big the value of k can be since we have to make sure that every level gets done using one pass only (k can only be as large as (M/B)).
  - In this case, we have an asymptotically optimal algorithm.
    - Number of passes = $\frac{\log(n/M)}{\log(M/B)} + 1$
    - Number of I/O's = $\frac{n}{B}\left(\frac{\log(n/M)}{\log(M/B)} + 1\right)$.