# CSE 4502/5717 Big Data Analytics – Spring 2018
# Lecture 3: January 29, 2018

## Parallel Algorithms: Prefix Computation

- **Input:** A sequence $X = k_1, k_2, \ldots, k_n$ of elements from a domain $\Sigma$. $\oplus$ is a binary, associative, and unit operation defined on $\Sigma$. Recall that an operation $\oplus$ on $\Sigma$ is associative if for any three elements $x, y, z$ in $\Sigma$, the following holds: $x \oplus (y \oplus z) = (x \oplus y) \oplus z = x \oplus y \oplus z$.
- **Output:** $k_1, k_1 \oplus k_2, k_1 \oplus k_2 \oplus k_3, \ldots, k_1 \oplus k_2 \oplus \cdots k_n$.

## A Divide-and-conquer Algorithm for Prefix Computation

- We will first discuss a divide-and-conquer parallel algorithm that employs $n$ CREW PRAM processors.
- Step 0) We partition the input into two equal halves: $X_1 = k_1, k_2, \ldots, k_{n/2}$ and $X_2 = k_{(n/2)+1}, k_{(n/2)+2}, \ldots, k_n$.
- Step 1) $\frac{n}{2}$ processors recursively perform a prefix computation on $X_1$. Let the output be $k'_1, k'_2, \ldots, k'_{n/2}$; At the same time the other $\frac{n}{2}$ processors recursively perform a prefix computation on $X_2$. Let the output be $k'_{(n/2)+1}, k'_{(n/2)+2}, \ldots, k'_n$.
- Note that $k'_1, k'_2, \ldots, k'_{n/2}$ is indeed the first half of the prefix outputs for $X$. Thus we can output these without any modifications.
- Step 3) We can modify $k'_{(n/2)+1}, k'_{(n/2)+2}, \ldots, k'_n$ by pre-adding $k'_{n/2}$ to every element. (Here the word 'adding' refers to the operator $\oplus$). The modified values will be the second half of the prefix outputs for $X$. This modification can be done in $O(1)$ time using $\frac{n}{2}$ CREW PRAM processors.

**Run time analysis:** Like for any recursive algorithm, we have to write a recurrence relation for the run time, and solve it. Let $T(n)$ be the run time of the above algorithm on any input of size $n$, where the number of processors used is $n$.

Then we get the following recurrence relation: $T(n) = T(n/2) + O(1)$ which solves to $T(n) = O(\log n)$. As a result, we get the following Lemma.

**Lemma 1:** *We can solve the prefix computation problem on any input of size $n$ in $O(\log n)$ time using $n$ CREW PRAM processors.* $\square$

# An Optimal Prefix Computation Algorithm

- We can get an optimal prefix computation algorithm using a technique due to Richard Brent. The idea is to reduce the input size sufficiently, employ a nonoptimal algorithm to solve the problem on the reduced input, and use these results to obtain the results for the original input. Let $P = \frac{n}{\log n}$ and let $X = k_1, k_2, \ldots, k_n$ be the input. A detailed description is given below.

0) Assign $\log n$ elements per processor. Specifically, assign the elements $k_{(i-1)\log n+1}, k_{(i-1)\log n+2}, \ldots, k_{i \log n}$ to processor $i$, for $1 \leq i \leq P$;
1) **for** $i = 1$ **to** $P$ **in parallel do**
2) Processor $i$ performs a prefix computation on its $\log n$ elements $k_{(i-1)\log n+1}, k_{(i-1)\log n+2}, \ldots, k_{i \log n}$ to get $k'_{(i-1)\log n+1}, k'_{(i-1)\log n+2}, \ldots, k'_{i \log n}$;
3) $P$ processors collective perform a prefix computation on $k'_{\log n}, k'_{2 \log n}, \ldots, k'_n$ to get $k''_{\log n}, k''_{2 \log n}, \ldots, k''_n$;
4) **for** $i = 2$ **to** $n$ **in parallel do**
5) Processor $i$ outputs $k''_{(i-1)\log n} \oplus k'_{(i-1)\log n+1}$, $k''_{(i-1)\log n} \oplus k'_{(i-1)\log n+2}, \ldots, k''_{(i-1)\log n} \oplus k'_{i \log n}$;

**Run time analysis:** Step 2 takes $O(\log n)$ time. In step 3 we have to perform a prefix computation on $\frac{n}{\log n}$ elements using $\frac{n}{\log n}$ processors. Using Lemma 1, we infer that Step 3 takes $O\left(\log\left(\frac{n}{\log n}\right)\right) = O(\log n)$ time. Step 5 takes $O(\log n)$ time. Thus the total run time of the algorithm is $O(\log n)$ resulting in the following Lemma.

**Lemma 2:** *We can solve the prefix computation problem on any input of size $n$ in $O(\log n)$ time using $\frac{n}{\log n}$ CREW PRAM processors.* $\square$

**Observation:** The above algorithm is asymptotically optimal since $S = n$ and the work done by the parallel algorithm is $O(n)$.

**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, etc.

- The devices of the list above are in increasing order in terms of the time it takes to access data from. 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.

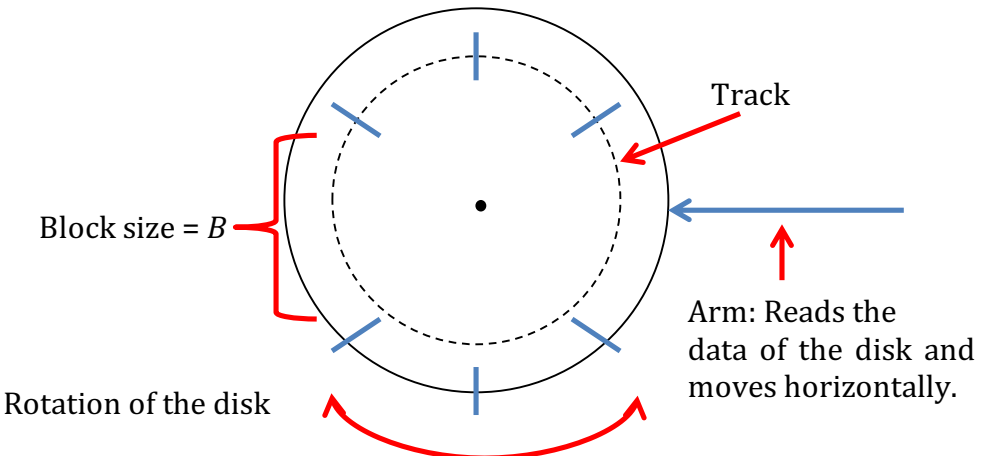
- **The time it takes to access one record in the various devices are:**
  - Registers –> Nanosecond ($10^{-9}$)
  - Core Memory -> Several Nanoseconds
  - Disk -> Several Milliseconds
  - SSDs -> Microseconds

- 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 operation. 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.

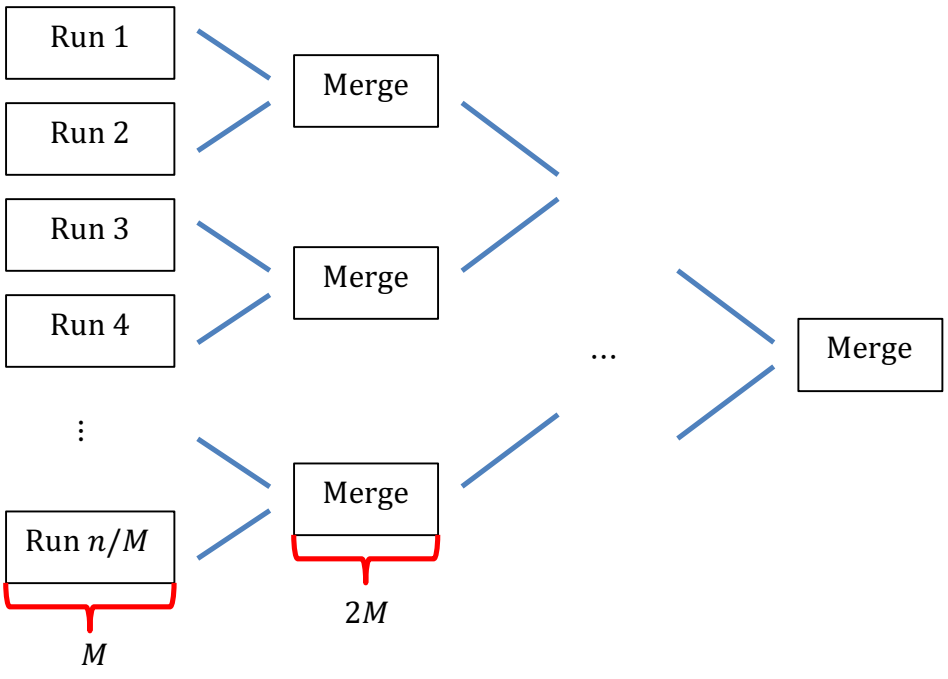
- Here *M* is the core memory size and *B* is the block size.

- **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:**

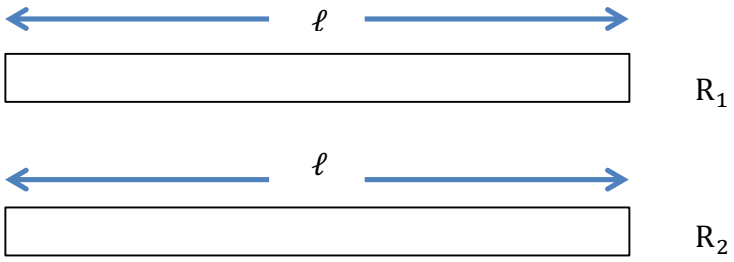
- **First attempt:**
  1) Do one pass through the data and form runs of length *M* each. A run is a sorted subsequence.
     - We have to merge these n/M runs.

  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 $\ell$ 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

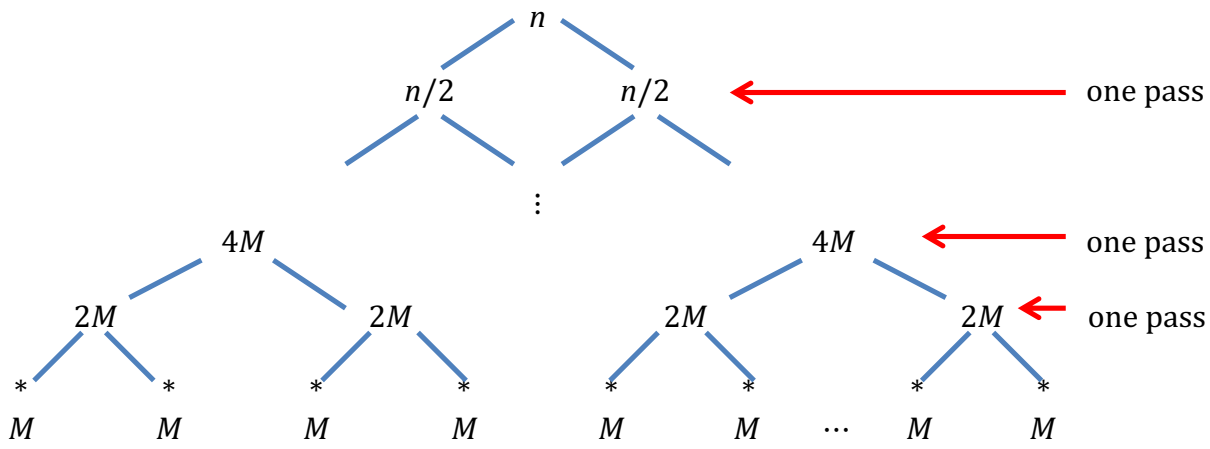
$R_2$: 9, 10, 11, 12, 13, 14, 15, 16

- **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:** $\frac{n}{B}\left(\log\left(\frac{n}{M}\right) + 1\right)$.

Note that this algorithm is not asymptotically optimal. We can modify this to make it optimal.