COMS E6998-9: Algorithms for Massive Data (Fall '25) Oct 15, 2025

Lecture 13: Large-Scale Models

Instructor: *Alex Andoni* Scribe: *Nicky Khorasani*

1 Introduction: Parallel/Distributed Models

Modern datasets often exceed the memory and compute of a single machine. We therefore consider algorithmic models that exploit multiple processors or many machines.

Efficiency goals: In large-scale settings we typically optimize some combination of:

- **Parallel time** T_p: number of synchronized steps until completion.
- **Work** W: total operations across all processors/machines.
- **Communication**: rounds of data exchange and total volume moved.
- **Space per machine** S (and total memory MS in distributed settings).

Compared to the classical RAM/Turing model (single processor, single memory), these models have trade-offs between computation and communication.

2 Model 1: PRAM (Parallel Random Access Machine)

2.1 Model description

In the PRAM model we have p identical processors (each with registers) and a **shared** memory. Computation proceeds in "time steps." In each step every processor can do one of the following:

1. Read from shared memory
2. Write to shared memory
3. Do local computation

Definition 1 (Parallel Time)**.** *The* parallel time T_p *is the number of synchronized PRAM steps required for the algorithm to finish with* p *processors.*

Definition 2 (Work)**.** *The* ***work*** *of a parallel algorithm is the total number of operations performed across all processors during its execution. Formally, if* T_p *is the parallel time then* $W = \sum_{t=1}^{T_p}(\#$ *active processors at step* $t)$*.*

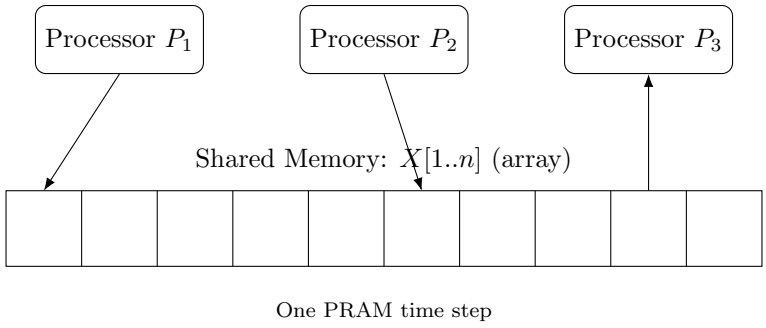

Figure 1: Three processors accessing, different locations of a shared array in one PRAM time step. P_1 and P_2 are writing and P_3 is reading.

2.2 Concurrent vs. exclusive access

There are different variants of PRAM that differ based on what types of memory accesses they allow.

- **EREW**: Exclusive Read, Exclusive Write (Cannot concurrently reads/write the same element in shared memory).
- **CRCW**: Concurrent Read, Concurrent Write. Concurrent writes are hard to define, need some sort of rule (e.g., arbitrary/priority/min/max).

Observation 3. *If a problem is solvable on a p-processor PRAM in time T_p, then a sequential simulation runs in time $O(W) = O(pT_p)$. Therefore the maximum speedup over the best sequential time is at most $O(p)$.*

2.3 Example: XOR of n bits on PRAM Model

Input: $x_1, \ldots, x_n \in \{0, 1\}$, output $\bigoplus_{i=1}^{n} x_i$.

Tree Reduction (EREW):
With $p = n$ processors, pairwise reduce along a binary tree:

1. Level 1: processors in parallel compute $x_{2j-1} \oplus x_{2j}$ for all j.
2. Level 2: reduce consecutive pairs of results, and so on.
3. After $\lceil \log_2 n \rceil$ levels, the root holds the XOR.

This achieves $T_p = O(\log n)$, work $W = O(n)$ (work-efficient).

Lemma 4. *Even allowing CRCW, computing XOR requires $\Omega(\frac{\log n}{\log \log n})$ parallel time.*

Historical note: PRAM model was developed before multi-processing really existed. It is a clean model, but real machines rarely provide uniform low-latency shared memory at scale so this assumption is unrealistic.

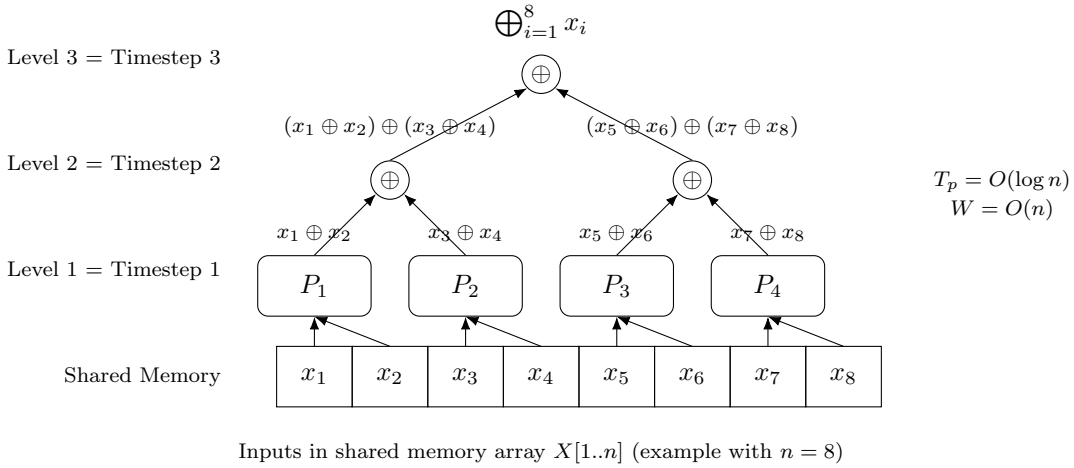

Figure 2: Tree reduction (EREW PRAM): at each level, processors compute pairwise XORs, *write results back to shared memory*, and the next level's processors *read* those results. Level 1 uses $n/2$ processors, Level 2 uses $n/4$, and so on, until one processor remains. $T_p = O(\log n)$ and $W = O(n)$.

3 Model 2: MPC (Massively Parallel Computation

3.1 Model description

This formulation was inspired by googles **Map Reduce System**. MPC models a cluster with M machines, each with local memory S. The input (of size n) is arbitrarily distributed across all machines. Typically $S \ll n$ and $MS \geq n$. Therefore, the dataset of size n cannot fit in the memory of a single machine, but it fits comfortably across all machines' total memory.

Computation happens in *rounds*. In each round we do the following:

1. **Local compute:** Each machine performs computation on its local data (usually linear time in its data).
2. **Shuffle:** Machines send their data to other machines, with the constraint that each machine receives at most S total data.

Performance measures.

- **Rounds** R: number of global compute/shuffle phases.
- **Per-round machine compute** $\leq S$ (i.e., linear compute on each machine)
- **Total work** $\approx R \cdot M \cdot S$.
- Typical regime: $MS = \Theta(n)$ and set $S = n^{\delta}$ for some constant $\delta \in (0, 1)$.

3.2 Relationship to PRAM

With $S = O(1)$ the model becomes a "message-passing" version of CREW-PRAM. One can show that such a PRAM algorithm can be made into a MPC algorithm with constant S with only $O(1)$ blowup in number of rounds.

Increasing S (e.g., $S = n^\delta$) reduces the number of rounds for many (but not all!) problems to $R = O(\log_S n) = O(1/\delta)$, highlighting a memory–communication trade-off.

4 Example: XOR in MPC

Partition the n bits across M machines (worst-case distribution allowed).

1. **Local step:** Each of the M machines XORs its local block into one bit.
2. **Shuffle/reduce:** Route those M bits to a designated machine responsible for aggregating results.

With $S = n^\delta$, a d-ary reduction yields

$$R \;=\; O(\log_S n) \;=\; O\left(\frac{1}{\delta}\right),$$

and the per-round machine remains $\leq S$. Therefore, the XOR can be computed in a constant number of rounds when each machine has polynomial (n^δ) memory. e.g. if $\delta = \frac{1}{2}$ then $S = \sqrt{n}$ and $R = O(2)$.

5 Example: Prefix Sum in MPC

Input: Pairs (a_i, i) for $i \in [n]$. Each pair is arbitrarily distributed across machines.

Goal: Compute prefix sums $\sigma_i = \sum_{j=1}^{i} a_j$, allowing output to be arbitrarily distributed.

Step 1: Sorting / Rebucket by index
First we redistribute the data so that each machine stores a contiguous range of indices.

- Sort or route each pair (a_i, i) by key i.
- After routing, each machine M_ℓ holds roughly S consecutive elements

$$M_1 : (a_1, \ldots, a_S), \quad M_2 : (a_{S+1}, \ldots, a_{2S}), \quad \ldots$$

- This can be done in a single round since we have the element indices.

Step 2: Up-sweep (computing partial sums):
Each machine locally computes the prefix sums of its own data and then sends its sum upward in a tree structure. This process repeats until we get to the root. See Figure 3. for the upsweep visualization.

- Each machine M_ℓ computes its local sum $T_\ell = \sum_{a_i \in M_\ell} a_i$.
- Parent machines in higher levels aggregate the totals received from the children machines to compute their subtree sum.

- This continues up the tree until the root holds the total sum $\sum_{i=1}^{n} a_i$.

Step 3: Down-sweep (propagating offsets):
Starting from the root with offset 0, offsets are pushed *down* the tree so that each machine learns the total sum of all elements preceding its block. See Figure 4 for a visualization.

- The root has offset 0 and sends it to its leftmost child.
- Each parent P maintains its current offset (the total sum of all elements before its subtree). Suppose P has children $C_1, \ldots, C_d$ with corresponding subtree totals $T_1, \ldots, T_d$ (computed during the up-sweep). Then:

$$\text{offset}(C_1) = \text{offset}(P), \quad \text{offset}(C_2) = \text{offset}(P) + T_1, \quad \text{offset}(C_3) = \text{offset}(P) + T_1 + T_2, \ \ldots$$

- Each leaf machine receives an offset equal to the prefix sum up to its first local element. It then adds this offset to all its local prefix sums to produce the correct global values.

In other words, every machine M_ℓ learns the total sum of all elements preceding its local block. This allows each machine to convert its local prefix sums into global prefix sums.

Overall complexity. The algorithm runs in $R = O(\log_S n)$ rounds with total work $O(MS \log_S n)$. At the end, every machine holds the correct global prefix sums for its indices.

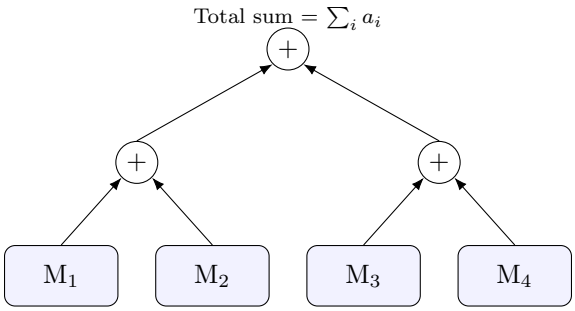

Figure 3: Up-sweep phase: local block sums are aggregated in a tree until the root holds the global total.

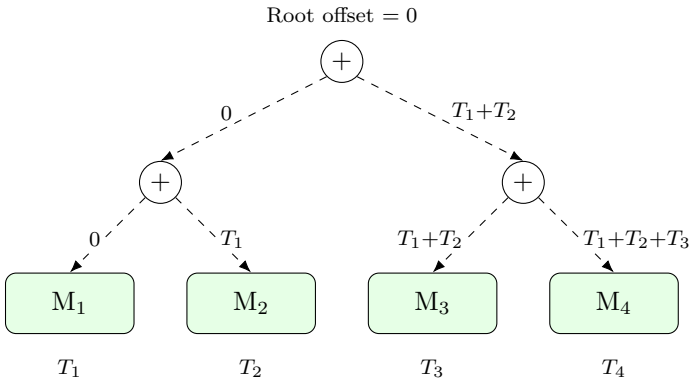

Figure 4: Down-sweep phase: each parent passes offsets to its children so every machine knows the total sum before its block. Leaf machines add these offsets to their local prefix sums to get global results.