CS260 – Lecture 6

Yan Gu

Algorithm Engineering
(aka. How to Write Fast Code)

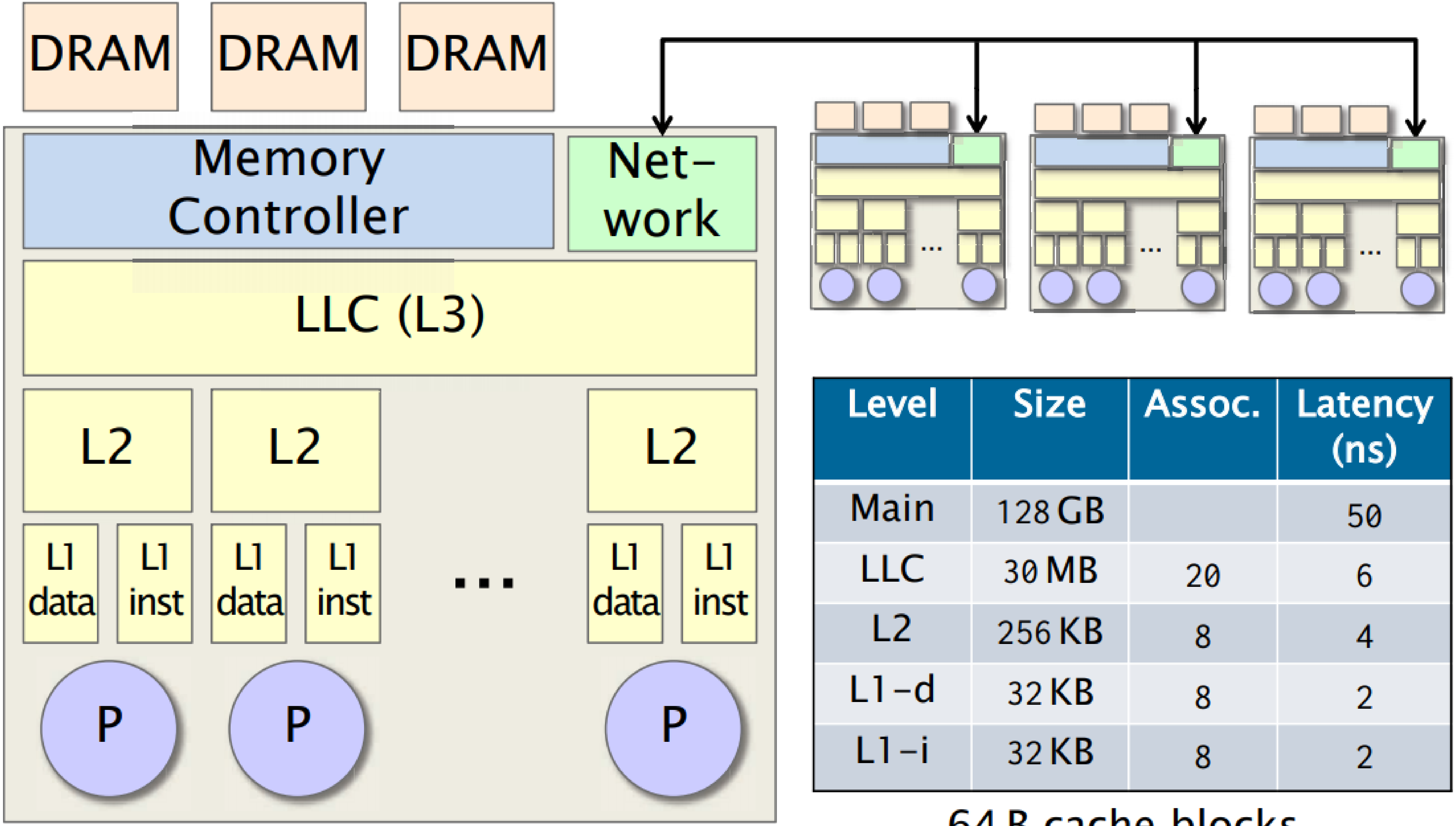
I/O Algorithms and Parallel Samplesort

CS260: Algorithm Engineering Lecture 6

The I/O Model

Sampling in Algorithm Design

Parallel Samplesort



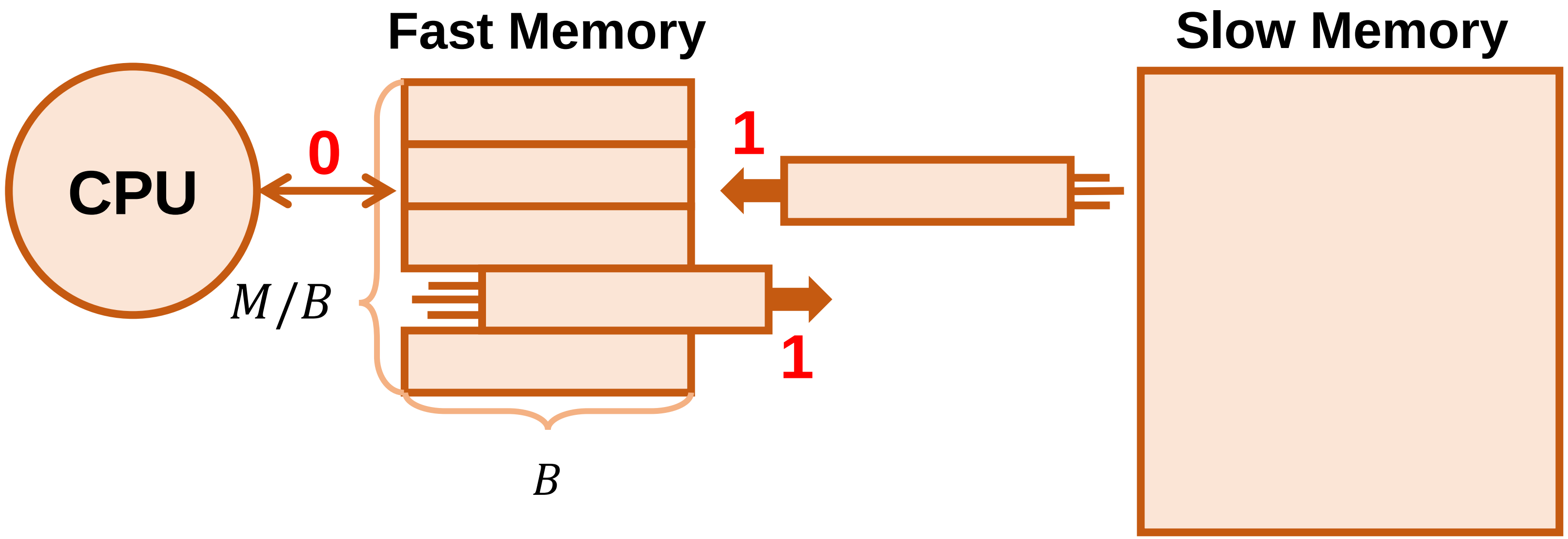
Level	Size	Assoc.	Latency (ns)
Main	128 GB		50
LLC	30 MB	20	6
L2	256 KB	8	4
L1-d	32 KB	8	2
L1-i	32 KB	8	2

64 B cache blocks

Last week – The I/O model

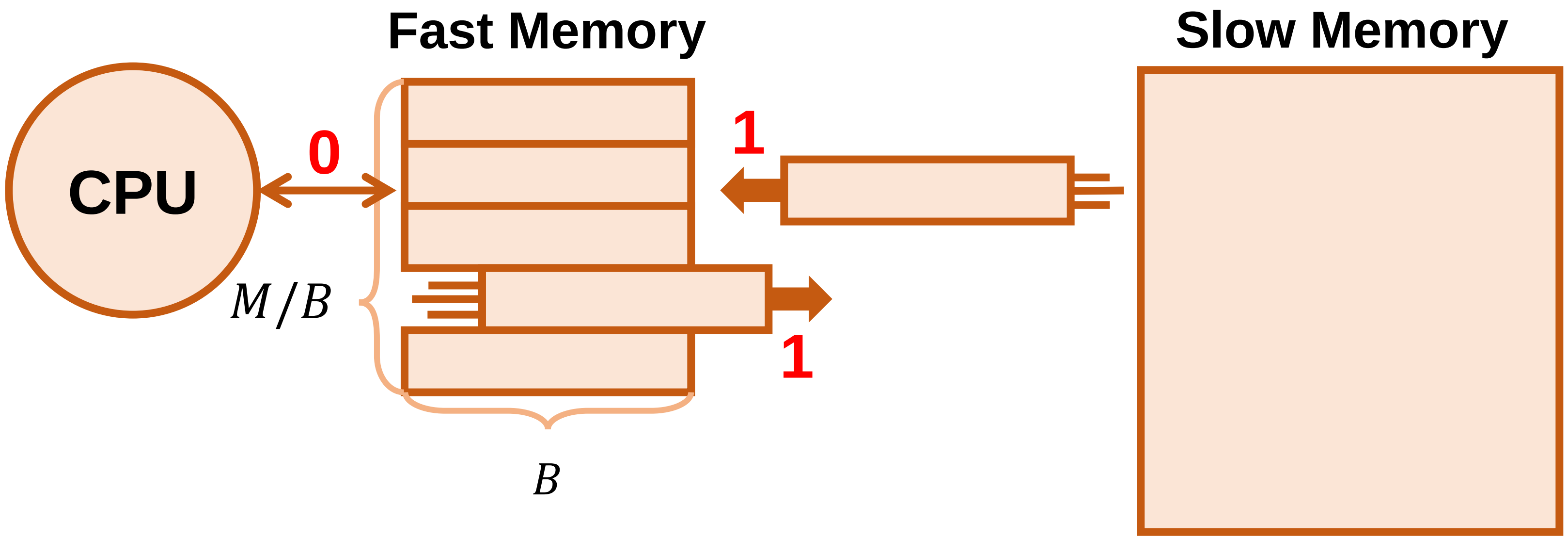
- The I/O model has two special *memory transfer* instructions:
  - **Read transfer**: load a block from slow memory
  - **Write transfer**: write a block to slow memory
- The complexity of an algorithm on the I/O model (I/O complexity) is measured by:

$$\#(\text{read transfers}) + \#(\text{write transfers})$$



Cache-Oblivious Algorithms

- Algorithms not parameterized by B or M
  - These algorithms are unaware of the parameters of the memory hierarchy
- Analyze in the *ideal cache* model — same as the I/O model except optimal replacement is assumed



CS260: Algorithm Engineering Lecture 6

The I/O Model

Sampling in Algorithm Design

Parallel Samplesort

Why Sampling?

- **Yan has an array $\{a_0, a_1, \dots, a_{n-1}\}$ such that $a_i = 0$ or 1, and Yan wants to know how many 0(s) in the array**

- **Scan, linear work, can be parallelized**
  - Sounds like a good idea?

Why Sampling?

- **Yan has an array $\{a_0, a_1, \dots, a_{n-1}\}$ and a function $f(\cdot)$ such that $f(a_i) = 0$ or 1, and Yan wants to know how many $f(a_i) = 0$**

Why Sampling?

- **Yan has an array $\{a_0, a_1, \ldots, a_{n-1}\}$ and n function $f_1(\cdot), \ldots, f_n(\cdot)$ such that $f_j(a_i) = 0$ or 1, and Yan wants to know how many $f_j(a_i) = 0$**

- **Takes quadratic work, does not work for reasonable input size**

- **Examples:**
  - Find the median m of a_i, $f_m(a_i) =$ "$a_i < m$", check if $\#(f_{a_j}(a_i) = 0)$ is $n/2$
  - Find a good pivot p in quicksort (e.g., $\frac{n}{4} \leq \#(f_p(a_i) = 0) \leq \frac{3n}{4}$)
  - Guarantee all sorts of properties in graph, geometry and other algorithms

Approximate Solution: Sampling

- **Yan has an array $\{a_0, a_1, \dots, a_{n-1}\}$ and n function $f(\cdot)$ such that $f(a_i) = 0$ or 1, and Yan wants to know how many $f(a_i) = 0$**

- **Uniformly randomly pick k elements, compute the $f(a_i) = 0$ case (denoted as k_0), and estimate by $\frac{n \cdot k_0}{k}$**
  - As long as k is sufficiently large, we are "confident" with our estimation
  - On the other hand, when k is small, the result can be random

- **When is the estimation good?**
- **What is "good"?**

Approximate Solution: Sampling

- **What is "good"?**
  - With high probability (informal): happens with probability $1 - n^{-c}$ for any constant $c > 0$
  - This is large when n is reasonably large, like $> 10^6$

- **When is the estimation good?**
  - Claim: when k_0 is $\Omega(\log n)$
  - How can reality off from the estimate?

Approximate Solution: Sampling

- **When is the estimation good?**
  - Claim: when k_0 is $\Omega(\log n)$
  - How can reality off from the estimate?
  - Assume there are z elements with $\boldsymbol{f(a_i) = 0}$, and we have k samples with k_0 hits. The expected #hits $\mathrm{E}[k_0] = kz/n$.
  - The probability that this is off by 100% (i.e., $k_0 > 2kz/n$) is $e^{-\frac{kz}{3n}}$

Chernoff bound: for n independent random variables in $\{0, 1\}$, let X be the sum, and $\mu = \mathrm{E}[X]$, then for any $0 \le \delta \le 1$,

$$\Pr(X \ge (1+\delta)\mu) \le e^{-\frac{\delta^2 \mu}{3}}$$

Approximate Solution: Sampling

- **When is the estimation good?**
  - Claim: when k_0 is $\Omega(\log n)$
  - How can reality off from the estimate?
  - Assume there are z elements with $\boldsymbol{f(a_i) = 0}$, and we have k samples with k_0 hits. The expected #hits $\mathrm{E}[k_0] = kz/n$.
  - The probability that this is off by 100% (i.e., $k_0 > 2kz/n$) is $e^{-\frac{kz}{3n}}$
  - Since $k_0 \approx kz/n$, $e^{-\frac{kz}{3n}}$ is n^{-c} when $k_0 = \Omega(\log n)$, because $e^{-\frac{kz}{3n}} \approx e^{-\frac{k_0}{3}} < e^{-c' \log_2 n} = n^{-c}$

Approximate Solution: Sampling

- **When is the estimation good?**
  - Claim: when k_0 is $\Omega(\log n)$
  - How can reality off from the estimate?
  - Assume there are z elements with $\boldsymbol{f(a_i) = 0}$, and we have k samples with k_0 hits. The expected #hits $\mathrm{E}[k_0] = kz/n$.
  - The probability that this is off by 1% (i.e., $k_0 > 1.01kz/n$) is $e^{-\frac{\delta^2 kz}{3n}}$
  - Since $k_0 \approx kz/n$, $e^{-\frac{\delta^2 kz}{3n}}$ is n^{-c} when $k_0 = \Omega(\log n)$, because $e^{-\frac{\delta^2 kz}{3n}} \approx e^{-\frac{k_0}{3 \cdot 100^2}} < e^{-c' \log_2 n} = n^{-c}$

Chernoff bound: for n independent random variables in $\{0, 1\}$, let X be the sum, and $\mu = \mathrm{E}[X]$, then for any $0 < \delta < 1$,

$$\Pr(X \geq (1 + \delta)\mu) \leq e^{-\frac{\delta^2 \mu}{3}}$$

Rule of Thumbs for Sampling

- **Example Applications:**
  - Find the median m of a_i, $f(a_i) = "a_i < m"$, check if $\#(f_{a_j}(a_i) = 0)$ is $n/2$
  - Find a good pivot p in quicksort (e.g., $\frac{n}{4} \leq \#(f_p(a_i) = 0) \leq \frac{3n}{4}$)
  - Guarantee all sorts of properties in graph, geometry and other algorithms

- **Take some samples! Uniformly randomly pick k elements, compute the $f(a_i) = 0$ case (denoted as k_0), and estimate by $\frac{n \cdot k_0}{k}$**
  - 4 sample hits gives you reasonable result
  - 20 sample hits gives you confident
  - 100 sample hits is sufficient!
  - Remember: only hits count

CS260: Algorithm Engineering Lecture 6

The I/O Model

Sampling in Algorithm Design

Parallel Samplesort

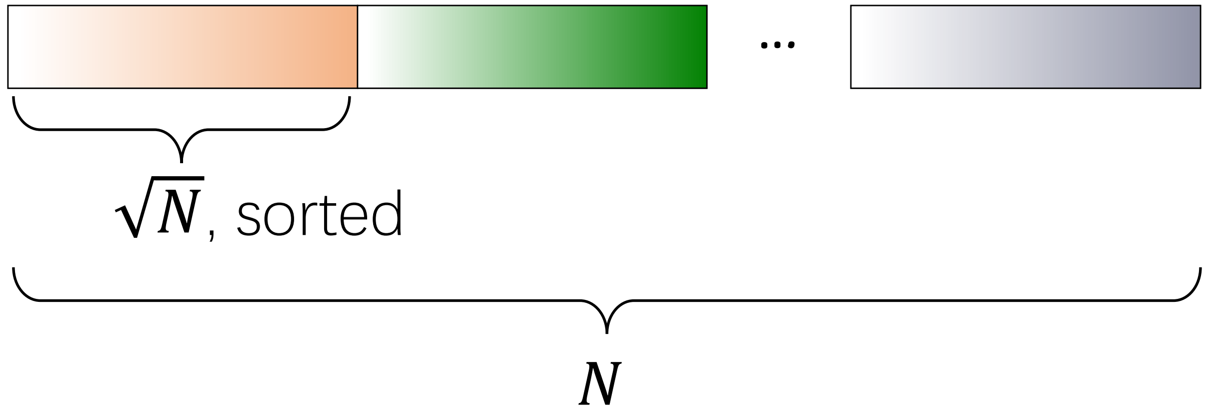
Parallel and I/O-efficient Sorting Algorithms

- **Classic sorting algorithms are easy to be parallelized**
  - Quicksort: find a "good" pivot, apply partition (filter) to find elements that are smaller and that are larger, and recurse
  - Mergesort: apply parallel merge for $\log_2 n$ rounds
  - But not I/O efficient since we need $\log_2 n$ rounds of global data movement
  - We now introduce samplesort, which is both highly in parallel and I/O efficient

Sample-sort outline

Analogous to multiway quicksort

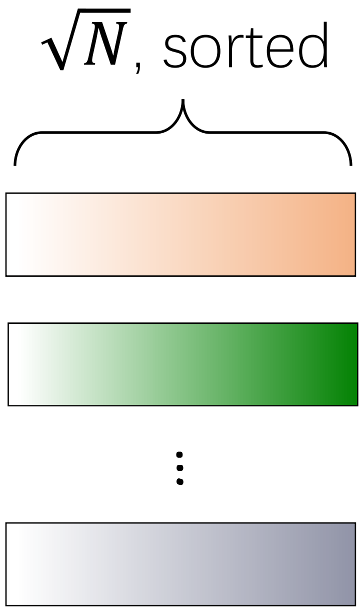
1. Split input array into $\sqrt{N}$ contiguous *subarrays* of size $\sqrt{N}$. Sort subarrays recursively



Sample-sort outline

Analogous to multiway quicksort

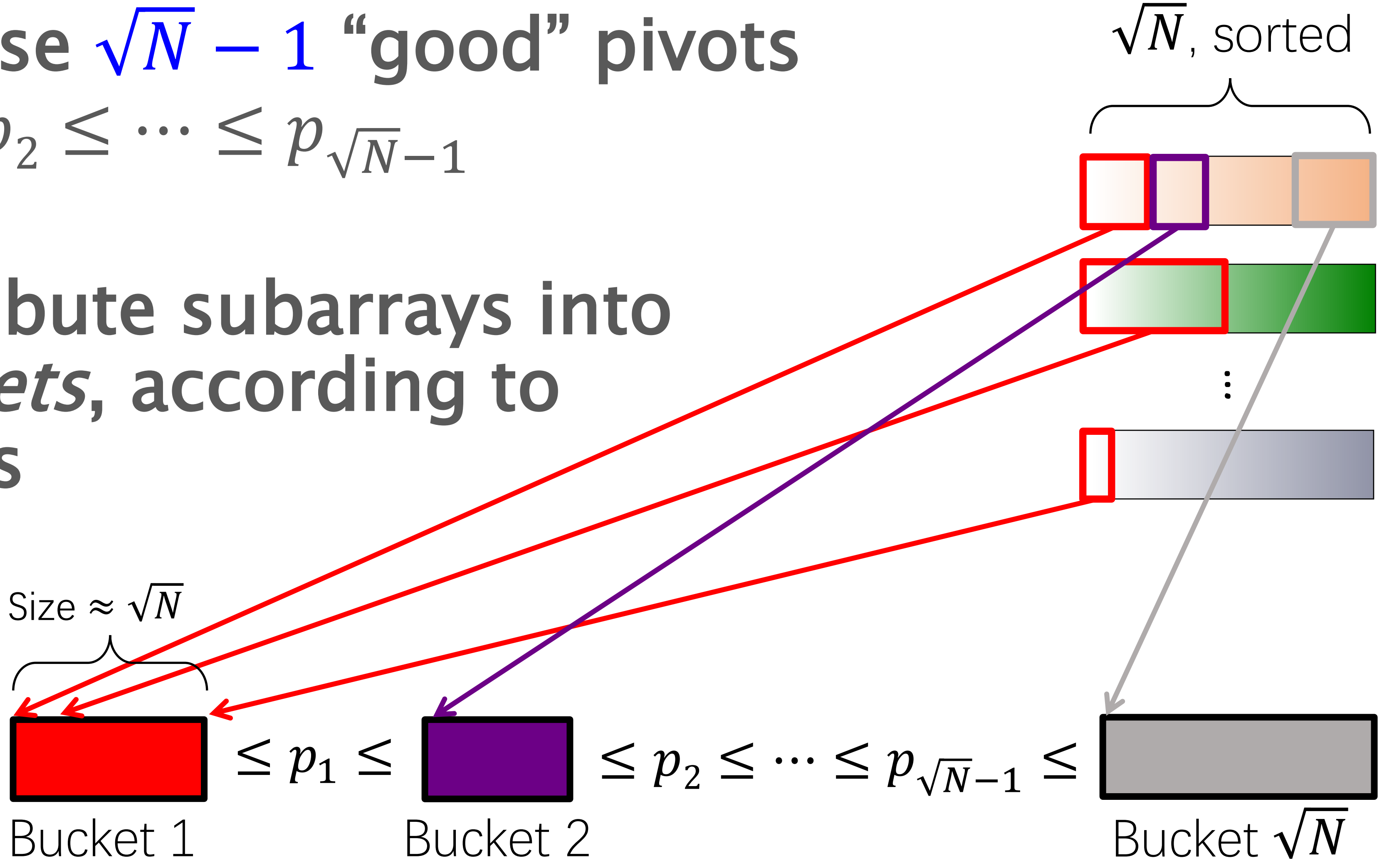
1. Split input array into $\sqrt{N}$ contiguous *subarrays* of size $\sqrt{N}$. Sort subarrays recursively (sequentially)



Sample-sort outline

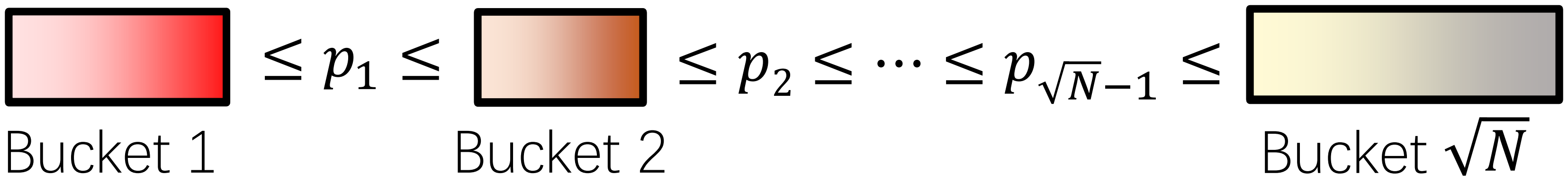
2. Choose $\sqrt{N} - 1$ "good" pivots
   $p_1 \le p_2 \le \cdots \le p_{\sqrt{N}-1}$

3. Distribute subarrays into *buckets*, according to pivots



Sample-sort outline

4. Recursively sort the buckets



5. Copy concatenated buckets back to input array

Choosing good pivots based on sampling

2. Choose $\sqrt{N}-1$ "good" pivots $p_1 \le p_2 \le \cdots \le p_{\sqrt{N}-1}$

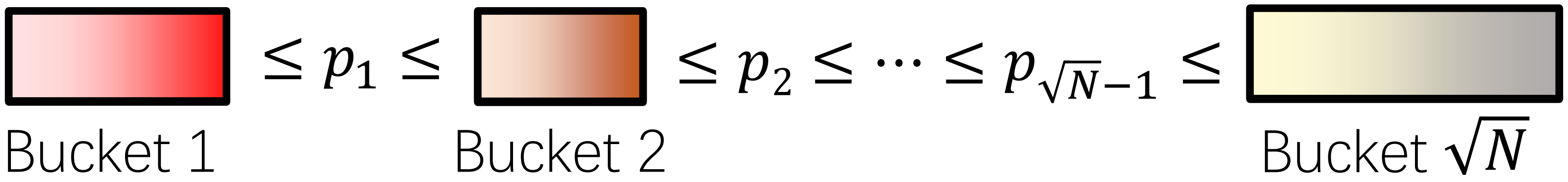
Can be achieved by randomly pick $c\sqrt{N}\log N$ random samples, sort them and pick the every $(c\log N)$–th element

This step is fast

Sequential local sorts (e.g., call stl::sort)

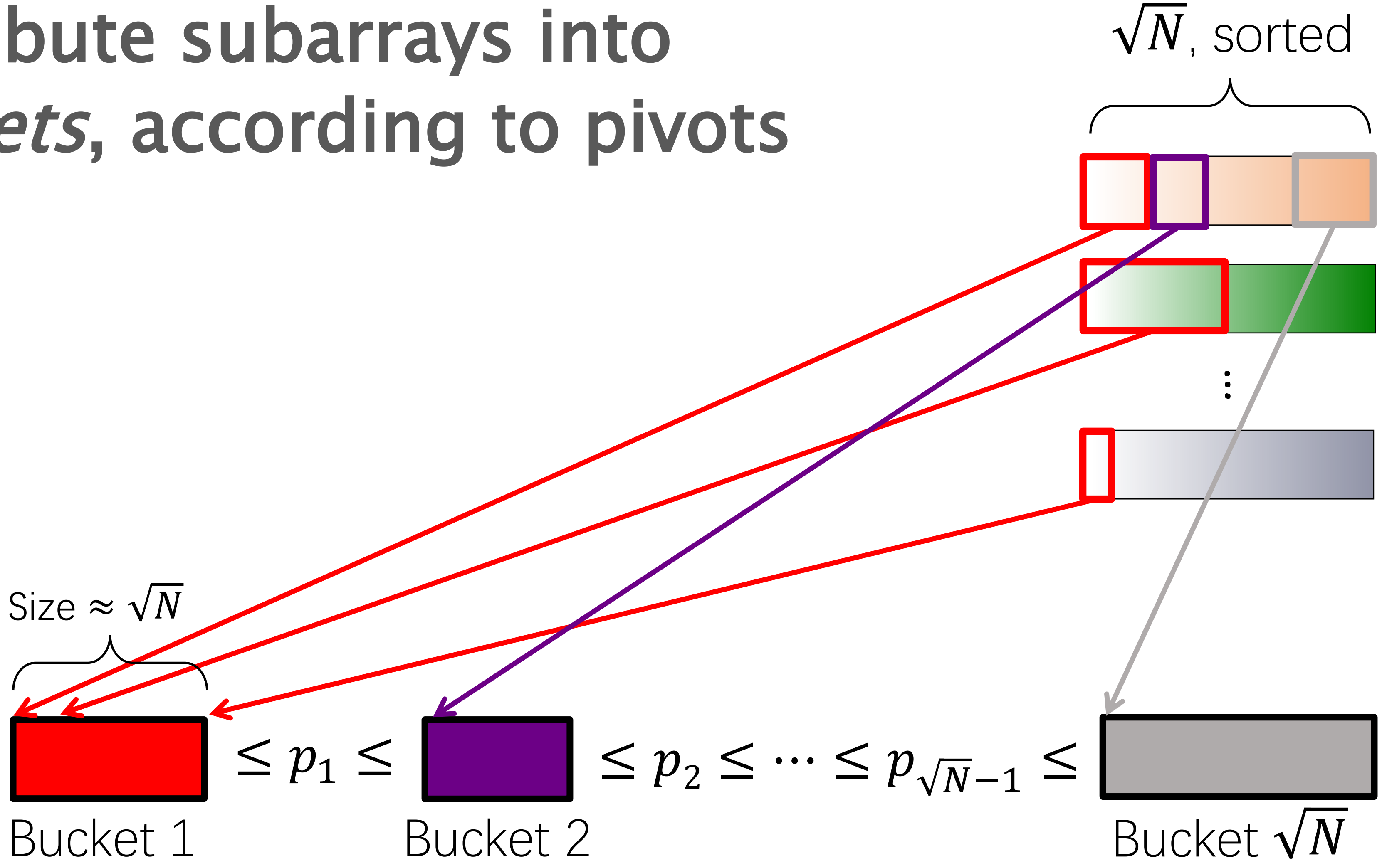
1. Split input array into $\sqrt{N}$ contiguous *subarrays* of size $\sqrt{N}$. Sort subarrays recursively (sequentially)

4. Recursively sort the buckets (sequential)



Key Part: the Distribution Phase

3. Distribute subarrays into *buckets*, according to pivots



Key Part: the Distribution Phase

- For simplicity, assume $n = 16$, and the input is

$$[1,2,3,4,\quad 1,1,3,3,\quad 1,2,2,4,\quad 1,2,4,4]$$

- First, get the count for each subarray in each bucket

$$[1,1,1,1,\quad 2,0,2,0,\quad 1,2,0,1,\quad 1,1,0,2]$$

- Then, transpose the array and scan to compute the offsets

$$[1,2,1,1,\quad 1,0,2,1,\quad 1,2,0,0,\quad 1,0,1,2]$$

$$[0,1,3,4,\quad 5,6,6,8,\quad 9,10,12,12,\quad 12,13,13,14]$$

- Lastly, move each element to the corresponding bucket

$$[1,1,1,1,1,2,2,2,2,3,3,3,4,4,4,4]$$

Additional Details Left for You

- **How to decide the count of each bucket in each subarray**
  - Hint: use a (sequential) merge algorithm
- **How to transpose the array for counts and write the elements to buckets I/O efficiently**
  - Hint: use divide-and-conquer
- **Find the best #pivots and #subarrays**
  - How does #pivots and #subarrays affect performance?

Samplesort is I/O-efficient

- **Only need two rounds of global data accesses**
  - For input size n between 10 million and 100 billion
- **In the midterm project, you can choose to implement this algorithm and engineer the performance**
  - This is harder than matrix multiplication, but easier than semisort
  - Expected score is 100%
- **Discussion: what is the work for samplesort? And what about depth?**

Next lecture: Semisort

- https://www.cs.ucr.edu/~ygu/teaching/algeng/algeng.html

- https://ilearn.ucr.edu/webapps/blackboard/execute/announcement?method=search&context=course&course_id=_307782_1