Simple Randomized Mresgerot

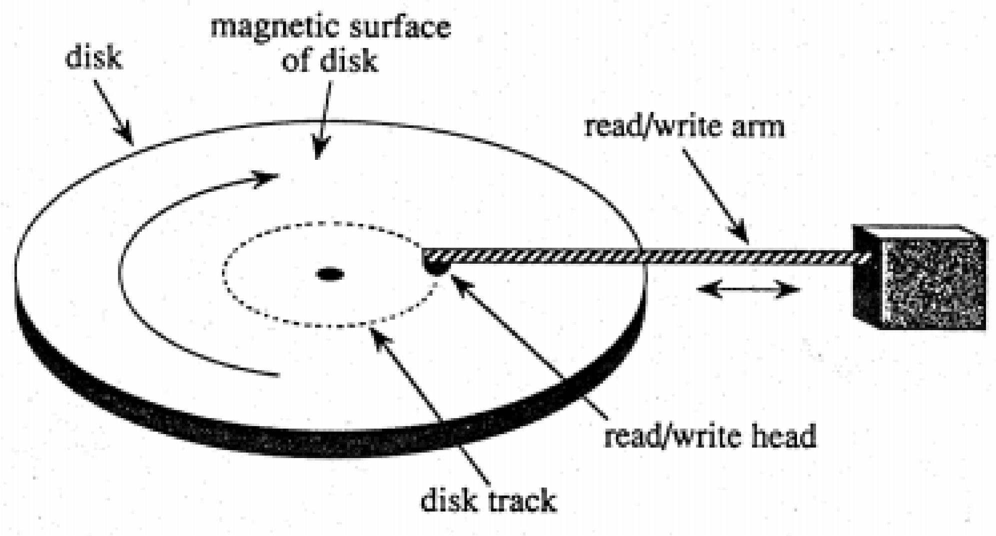
Jeff Vitter

Duke University
Department of Computer Science

Center for Geometric & Biological Computing
http://www.cs.duke.edu/CGBC/

EEF Summer School—July 2002

Magnetic Disk Drives as Secondary Memory



- ☆ I/O Crisis! 10^6 times slower access than registers.
- ☆ Time for rotation ≈ Time for seek.
- ☆ Amortize search time by large block transfer so that
  Time for rotation ≈ Time for seek ≈ Time to transfer data.
- ☆ Parallel disks.

Trends

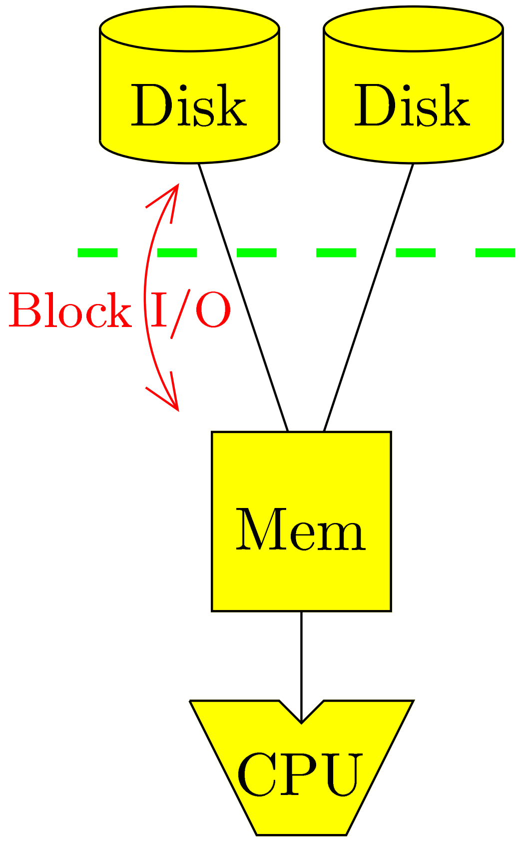
[Dahlin 96]:

Parameter	Yearly Improvement Rate
Disk Latency	10%
Disk Bandwidth	20%
Processor Speed	55%
RAM Bandwidth	40%
RAM Capacity/Cost	45%

- ☆ Performance gap is increasing.
- ☆ RAM Capacity/Cost doubling every 22–23 months, but users doubling data storage every 5 months. [AUS98]
- ☆ Users frequently reprocess data in entirety. [AUS98]
- ☆ I/O Bottleneck.

Parallel Disk Model

[Aggarwal & Vitter 88], [Vitter & Shriver 90, 94]



N = problem data size.

M = size of internal memory.

B = size of disk block.

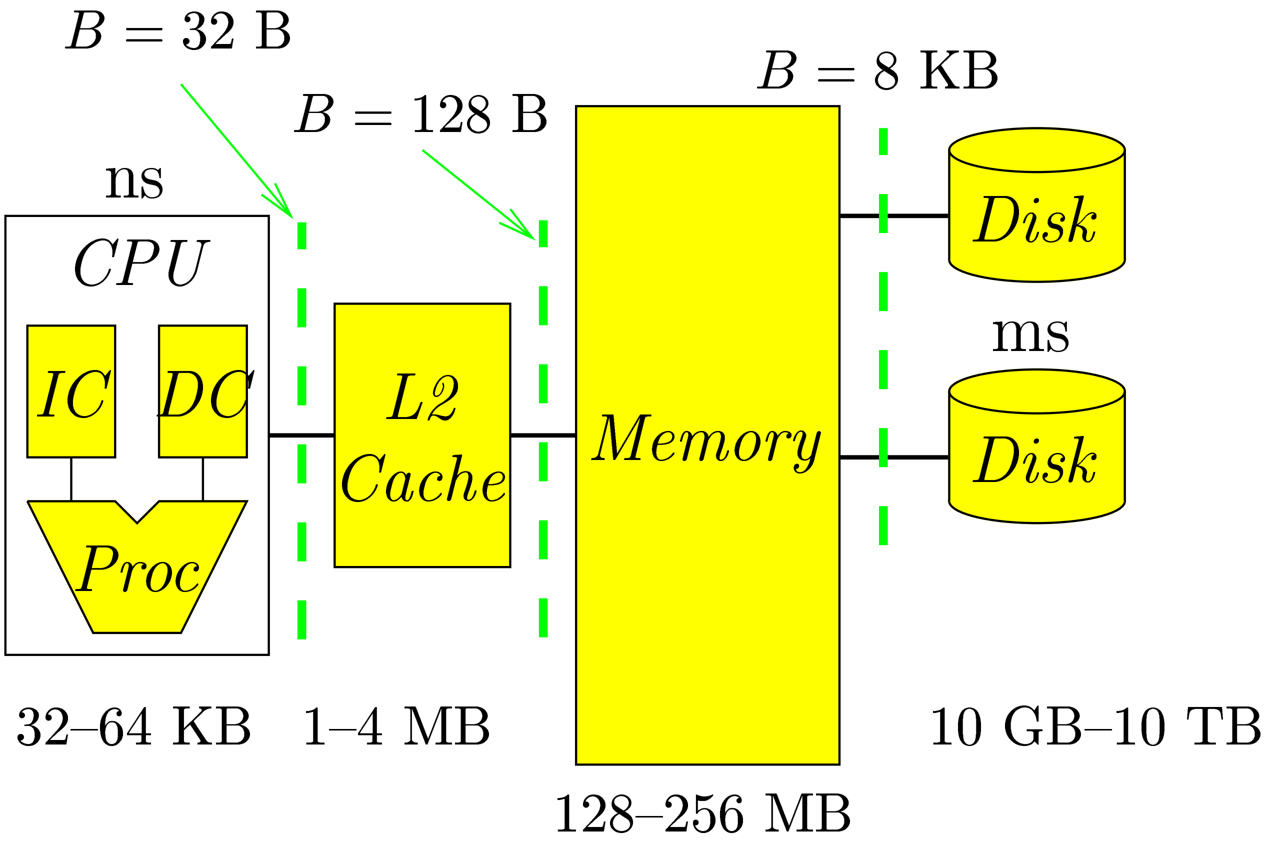
D = number of independent disks.

P = number of CPUs.

Notational convenience (in units of blocks):

$$n = \frac{N}{B}, \quad m = \frac{M}{B}.$$

A “Real” Machine



Outline of Talk

- ⋆ Single Disk Model, $D = 1$.
  - Lower Bound on sorting.
  - Single Disk Mergesort.
- ⋆ Parallel Disk Model, $D > 1$.
  - Difficulties of Parallelization.
  - Previous Approaches.
  - Simple Randomized Mergesort (SRM) and its analysis.
- ⋆ Implementation of SRM.

Fundamental Bounds

- Batched problems [AV88], [VS90,VS94]:
  - Scanning (touch problem): $\Theta\left(\frac{N}{DB}\right) = \Theta\left(\frac{n}{D}\right)$
  - Sorting:
    $$\Theta\left(\frac{N}{DB}\frac{\log \frac{N}{B}}{\log \frac{M}{B}}\right) = \Theta\left(\frac{N}{DB}\log_{M/B}\frac{N}{B}\right) = \Theta\left(\frac{n}{D}\log_m n\right)$$
  - Permuting: $\Theta\left(\min\left\{\frac{N}{D}, \frac{n}{D}\log_m n\right\}\right)$
- Sorting is key subroutine for many problems [CGGTVV95], [AKL95], . . .
  - Graph problems $\asymp$ Permutation
  - Computational Geometry $\asymp$ Sorting
- Online problems:
  - Searching: $\Theta(\log_{DB} N + z)$

Disk Striping: $D = 5$ disks, block size $B = 2$

- Data Layout:

	$\mathcal{D}_0$	$\mathcal{D}_1$	$\mathcal{D}_2$	$\mathcal{D}_3$	$\mathcal{D}_4$
stripe 0	0 1	2 3	4 5	6 7	8 9
stripe 1	10 11	12 13	14 15	16 17	18 19
stripe 2	20 21	22 23	24 25	26 27	28 29
stripe 3	30 31	32 33	34 35	36 37	38 39

- Disk striping involves using the D disks in lock step as if thre is a logical block size of BD

  $\Longrightarrow$ Substitute $B \leftarrow DB$ in single-disk algorithm.

- Single-disk I/O bound $\Theta\left(\frac{N}{B}\log_{M/B}\frac{N}{B}\right)$ becomes

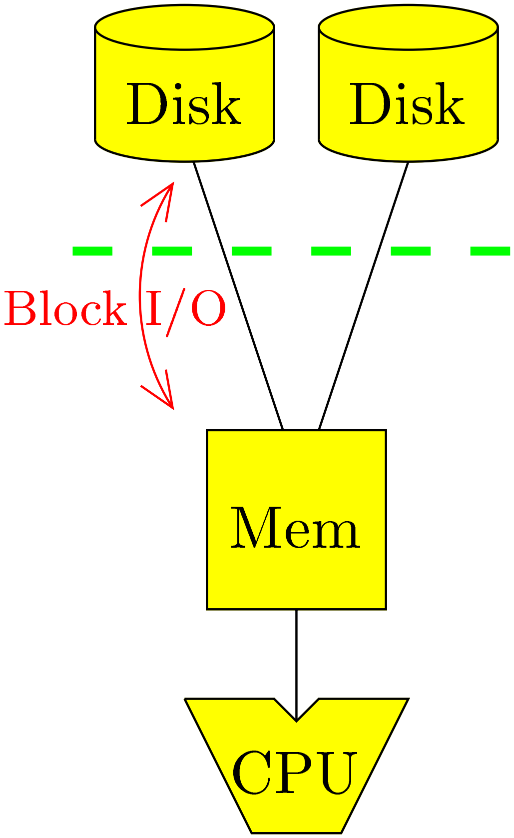
$$\Theta\left(\frac{N}{DB}\log_{M/DB}\frac{N}{DB}\right) = \Theta\left(\frac{n}{D}\log_{m/D}\frac{n}{D}\right)$$

- Ratio with optimal bound $\Theta\left(\frac{n}{D}\log_m n\right)$ is $\approx \frac{\log m}{\log \frac{m}{D}}$ when $D \approx m$.

- To get an optimal sorting algorithm, *use disks independently!*

Parallel Disk Model

[Aggarwal & Vitter 88], [Vitter & Shriver 90, 94]



☆ D = number of independent disks.

☆ Goal:

Design computation to transfer $\Theta(D)$ blocks in each I/O (one per disk).

☆ Optimal Sorting: $\Theta\left(\frac{n}{D}\log_m n\right)$ I/Os.

☆ Desired Sort Performance:

$$\begin{aligned} \#\ \text{passes} &= \Theta\left(\log_m n\right); \\ \#\text{I/Os per pass} &= \Theta\left(\frac{n}{D}\right). \end{aligned}$$

Distribution Sort with D Disks

- ☆ Distribution (bucket) sort
  - Select $S = \Theta(m)$ or $\Theta(\sqrt{m}\,)$ partitioning elements that divide the file evenly into buckets.
  - Sort the buckets recursively.
  - Append together the sorted buckets.
- ☆ The number of levels of recursion is $\log_S n = \log_m n$.
- ☆ If each level of recursion uses $\Theta\left(\frac{N}{DB}\right) = \Theta\left(\frac{n}{D}\right)$ I/Os

$$\Longrightarrow \quad \#\ \text{I/Os} = O\left(\frac{n}{D}\log_m n\right).$$

- ☆ The partitioning into buckets is done in an online manner as the data is streaming through memory: Whenever a bucket's buffer fills, it is written to disk.
- ☆ Difficulty is to store each bucket evenly across the disks, given that the blocks of each bucket are formed online.

Do We "Stripe" Buckets Contiguously on the Disks?

We have a choice of how to organize the distribution sort:

1. Make each bucket occupy contiguous "stripes" on the disks.
   For example, put bucket 1 on
   disk 1 track 1, disk 2 track 1, disk 3 track 1, ...
   Then when we output buffers during the bucketing process, each block will have a predefined disk to be written to.
   $\Longrightarrow$ possible bottleneck on a disk during write I/O.

2. Output the D blocks always in $O(1)$ I/Os, but a bucket may end up with more blocks on one disk than on other disks.
   $\Longrightarrow$ a bucket may end up unevenly distributed on the disks,
   leading to nonoptimal read pass in next level of recursion.

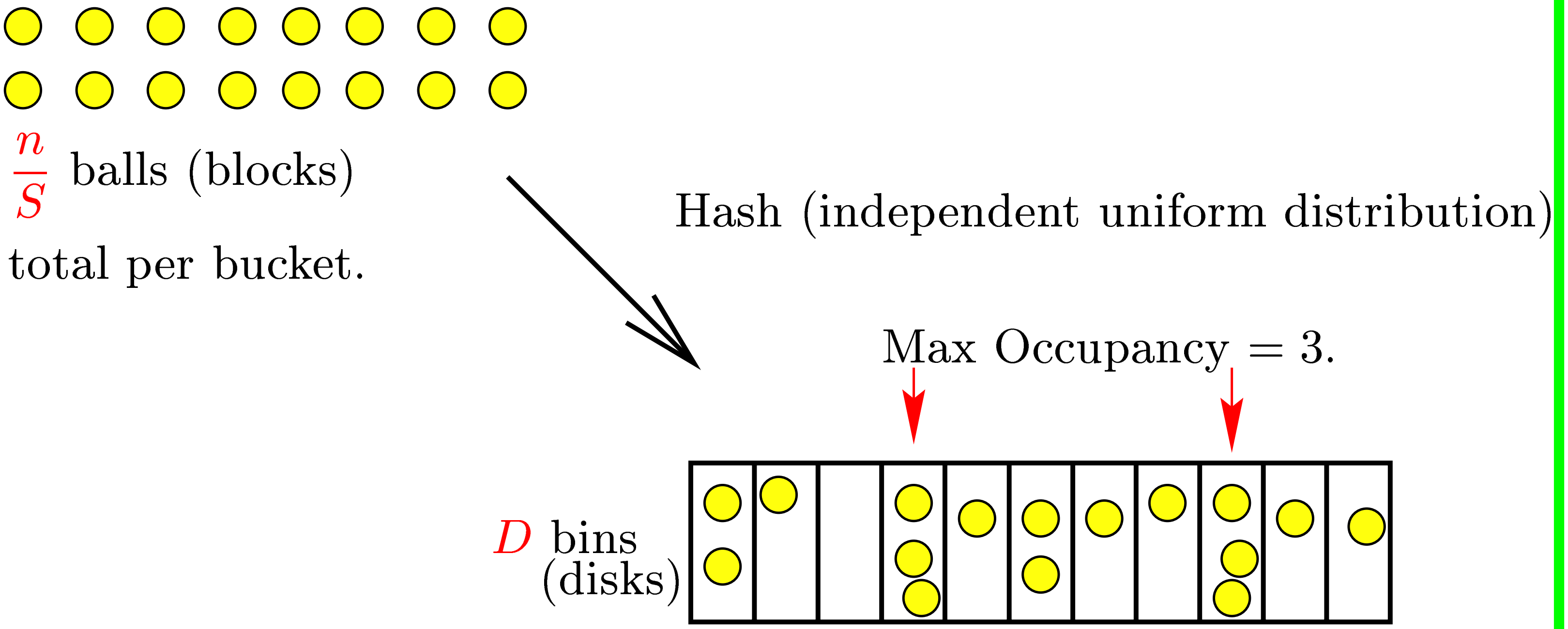
Hybrid versions are possible, such as when each bucket resides on contiguous stripes, but the order in which the blocks in each stripe are written can be arbitrary.

Early methods looked at Method 2 for distribution sort.
Newer methods concentrate on Method 1.

Bucket Sort [VS94]: Phase 1

Method 2 style

- ☆ If N is large or $\frac{M}{DB} > \log D$, then random assignment to disks works well.
- ☆ S simultaneous load balancing problems (one per bucket).



$\frac{n}{S}$ balls (blocks) total per bucket.

Hash (independent uniform distribution)

Max Occupancy = 3.

D bins (disks)

Bucket Sort [VS94]: Phase 2

- ☆ If N (and S) are small and $DB \approx M$ (so that random assignment is not "balanced"), a "typical" memoryload contains more than $S \log S$ blocks (and is therefore well-balanced among the S buckets.)
- ☆ Get a "typical" memoryload by permuting each memoryload and then shuffling the memoryloads in a single pass to mix them up randomly.
- ☆ Output each memoryload by a round-robin placement (perfect shuffle) of the S buckets onto the D disks.

BalanceSort [NV93]

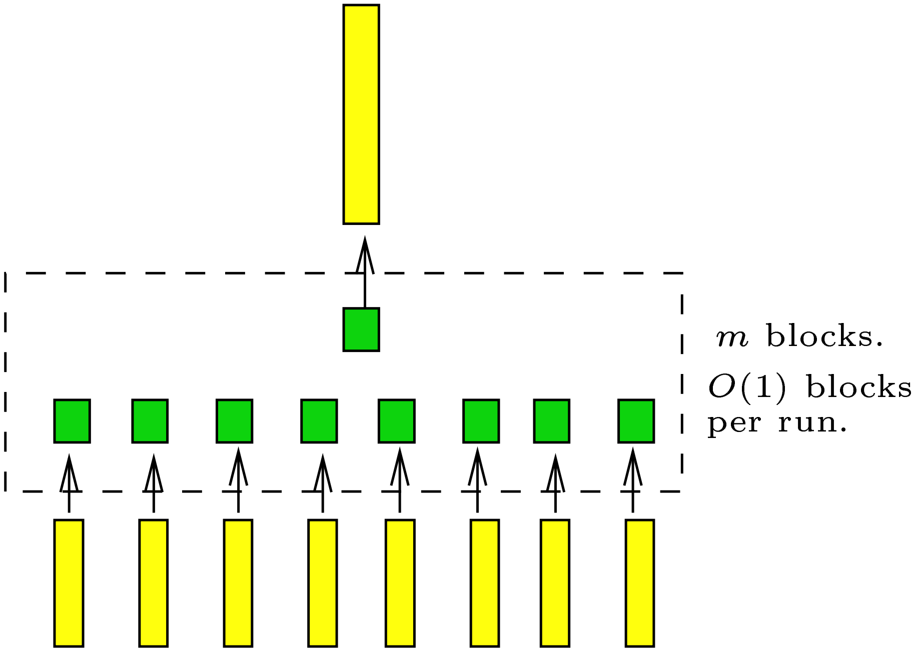
Deterministic version of BucketSort.

- ☆ Online tracking of bucket distribution on disks.
- ☆ Let num_b = # items in bucket b processed so far.
- ☆ Let $num_b(d)$ = # items in bucket b written to disk d, i.e., $num_b = \sum_{1 \le d \le D} num_b(d)$.
- ☆ Maintain invariant that the $\left\lfloor \frac{D}{2} \right\rfloor$ largest values of $num_b(1)$, $num_b(2)$, …, $num_b(D)$ differ by at most 1.

$$\Longrightarrow \quad num_b(d) \le 2\frac{num_b}{D}, \quad \text{for each bucket } b.$$

Back to Method 1

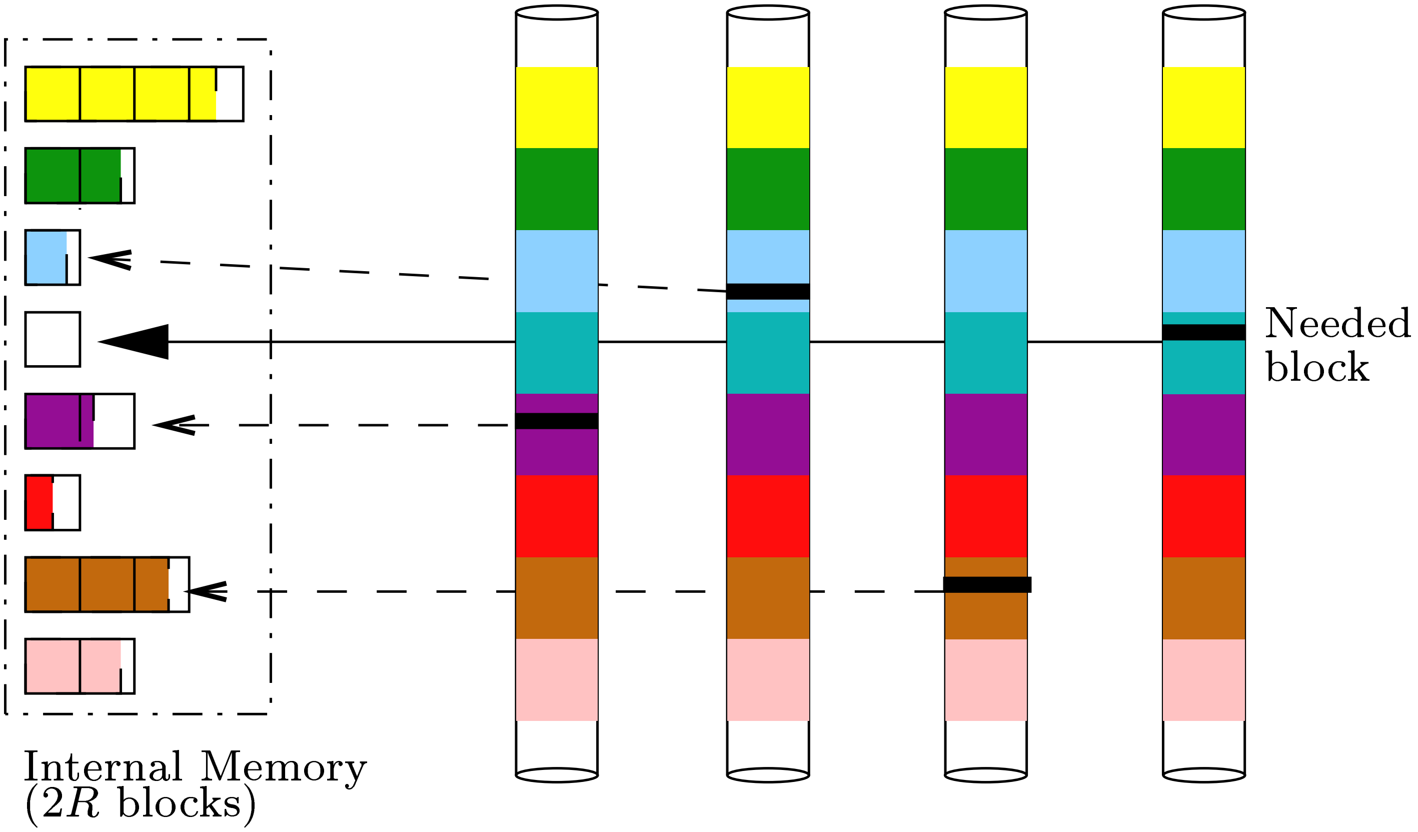
Let's concentrate for rest of talk on Method 1: Each bucket or run resides on "striped" predefined positions on the disks.



Merge $\Theta(m)$ runs together at a time.

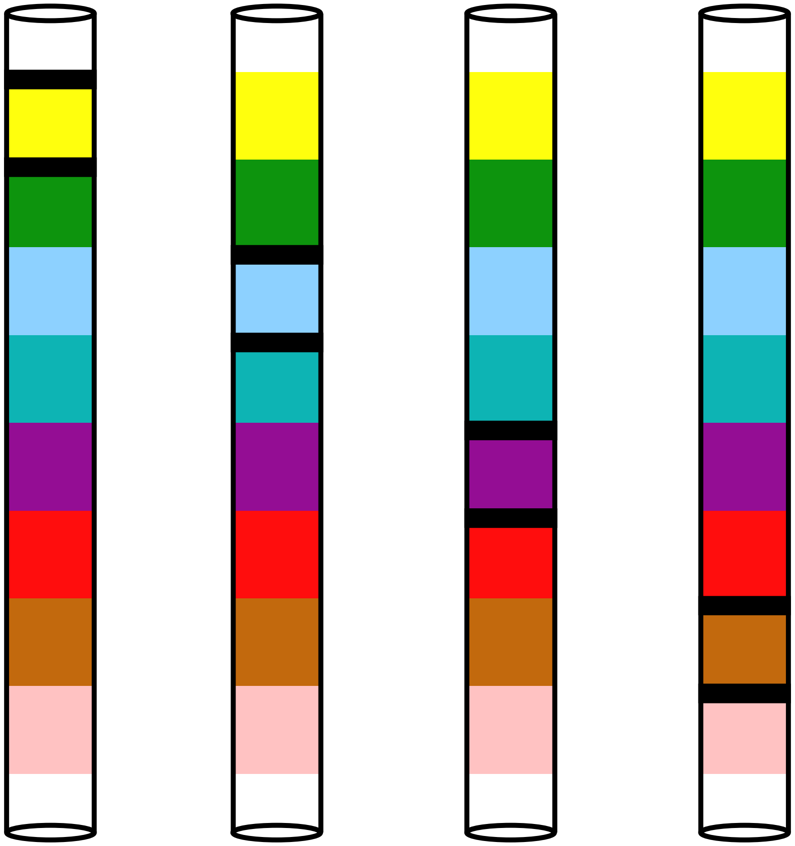
- ☆ Sort individual memoryloads to create runs of M records each.
- ☆ Merge $R = \Theta(m)$ or $\Theta(\sqrt{m}\,)$ runs at a time
  $\Longrightarrow \lceil \log_m n \rceil$ merge passes
- ☆ If each merge pass takes $O(n/D)$ I/Os
  $\Longrightarrow$ Total of $O\left(\frac{n}{D} \log_m n\right)$ I/Os for sorting (optimal).

Difficulty of merging $R = \Theta(m)$ runs on D disks



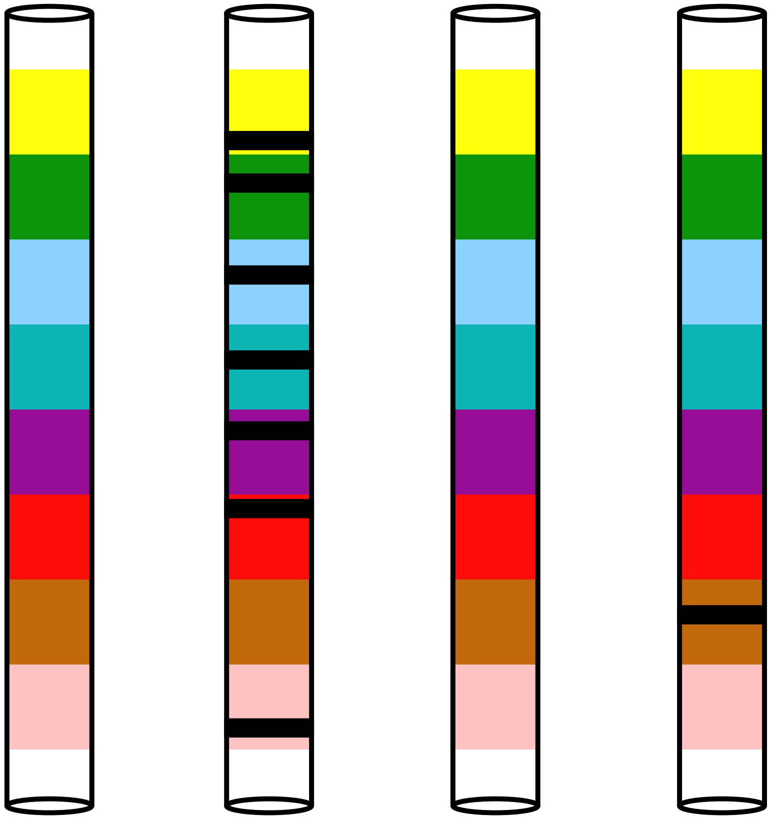
- ☆ Each read: One needed block, $D-1$ prefetched blocks for future use.
- ☆ Bounded memory size inhibits prefetching.
- ☆ Performance critical question: How many reads required to bring in the "next" $R = \Theta(m)$ blocks? Ideally, R/D.

Staggered Layout: $R = 8$, $D = 4$.



- ⋆ Each run is striped, but the starting disk of the runs are staggered.
- ⋆ Initially, # I/Os need to read in the next R leading blocks
  $=$ Max Occupancy (of leading blocks) on any disk
  $= R/D$, as desired.
- ⋆ But balance can quickly deteriorate.

$R = 8$, $D = 4$: Imbalance can set in.



Maximum Occupancy (of leading blocks on disks) is $7 \gg R/D$

$\Longrightarrow$ # I/Os needed to load 8 blocks = 7.

Gilbreath Principle

☆ Can achieve perfect balance for merging two runs, $R = 2$:

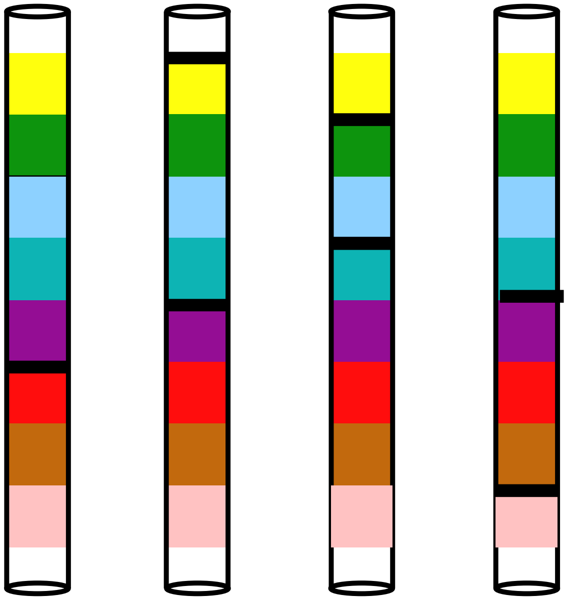
Run 1: A B C D
E F G H
I J K L
…

Run 2: D C B A (striped in reverse order)
H G F E
L K J I
…

☆ Reduces necessary buffer space by half.

☆ Cannot be generalized to $R > 2$.

Greed Sort [NV91]



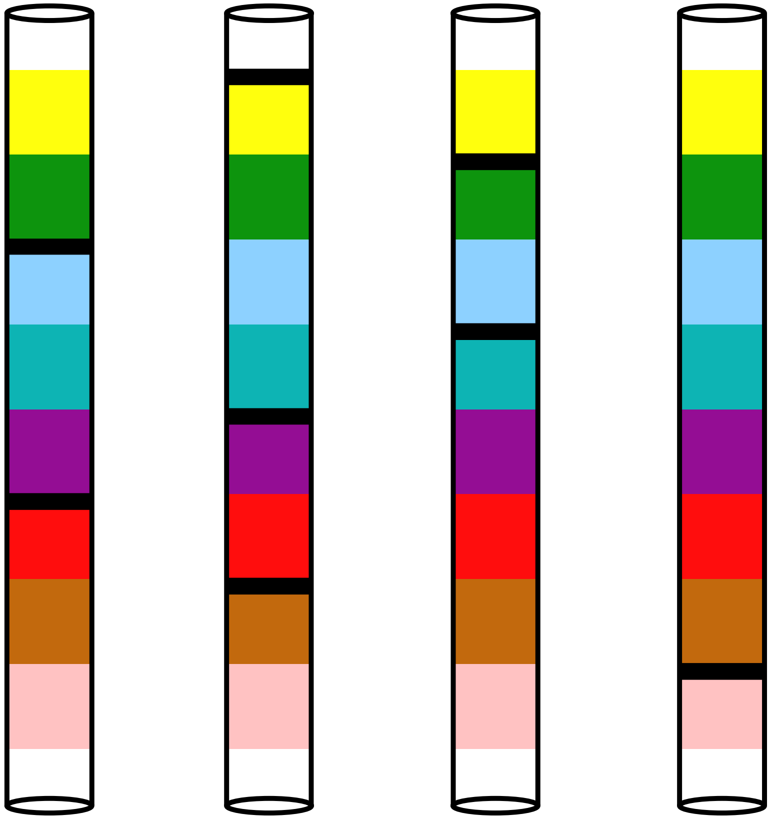
Overall structure of each merge pass:

1. Do approximate merge independently on each disk.
2. Interleave the "sorted" runs.
3. Use Columnsort to convert the approximately sorted output run into a totally ordered output run.

Merge procedure for each disk:

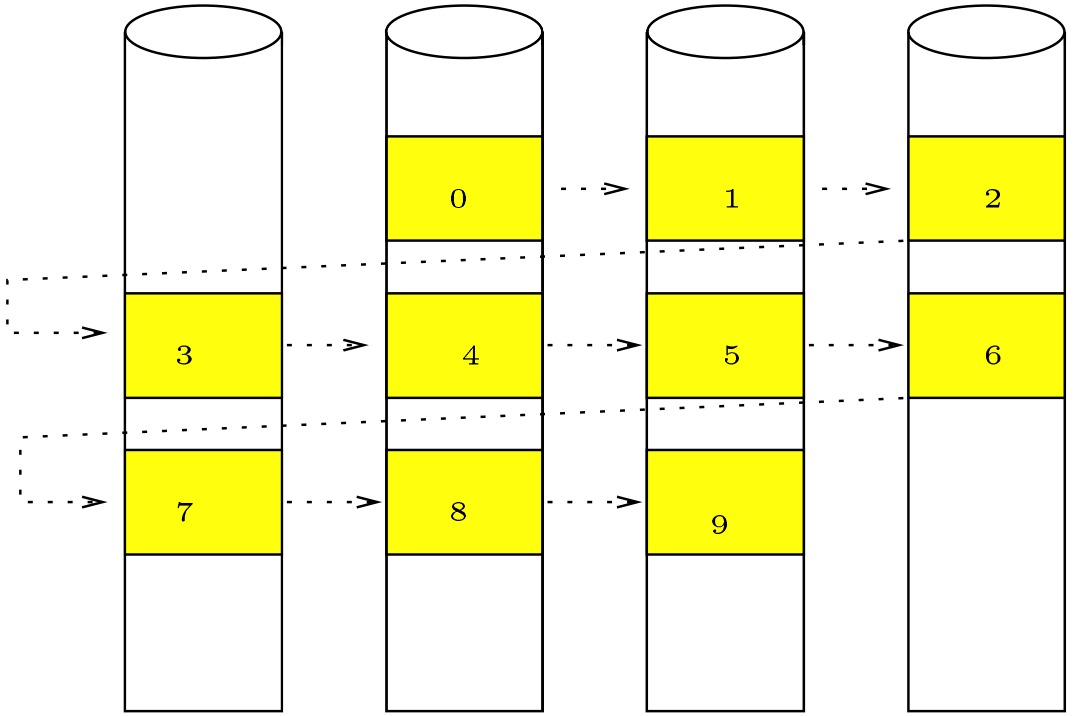
- ☆ Read the two blocks with smallest and smallest maximum items
- ☆ Output the smallest B items of the $2B$ items.

Simple Randomized Mergesort (SRM) [97,99]



- ⋆ Each run is striped starting at a randomly chosen disk.
- ⋆ *At any time*, the disk containing the leading block of any run is uniformly random.

Forecasting Information in an SRM run



Forecasting Information in the input blocks of a run:

Implanted in block i is the smallest key of block $i + D$.

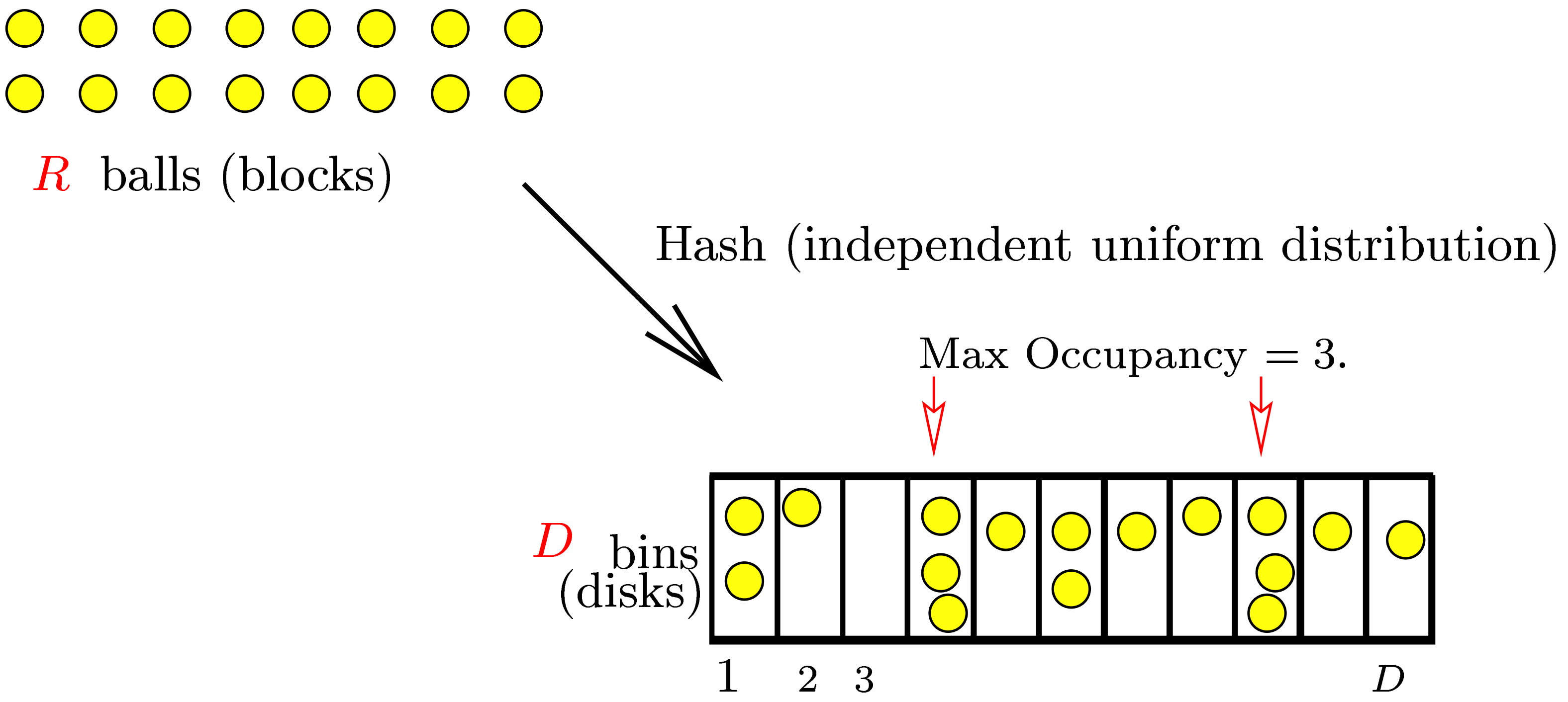
When block i is in memory, the time at which the merge needs block $i + D$ can be predicted.

SRM's Greedy Approach: Forecast and Flush (FF)

- Writes occur at full D-disk parallelism.
- SRM implants forecasting info in each input block.
- FF buffer management: (greedy approach)
  - If # Free Blocks is $D - f$, Flush f "largest" blocks.
  - Forecast the "smallest" block from each disk.
  - Read in the "smallest" block from each disk.
- Analysis: If $E[\mathrm{MaxOcc}_{SRM}]$ is the average Max Occupancy of the next R blocks, let's look at how many I/Os SRM uses to retrieve them:

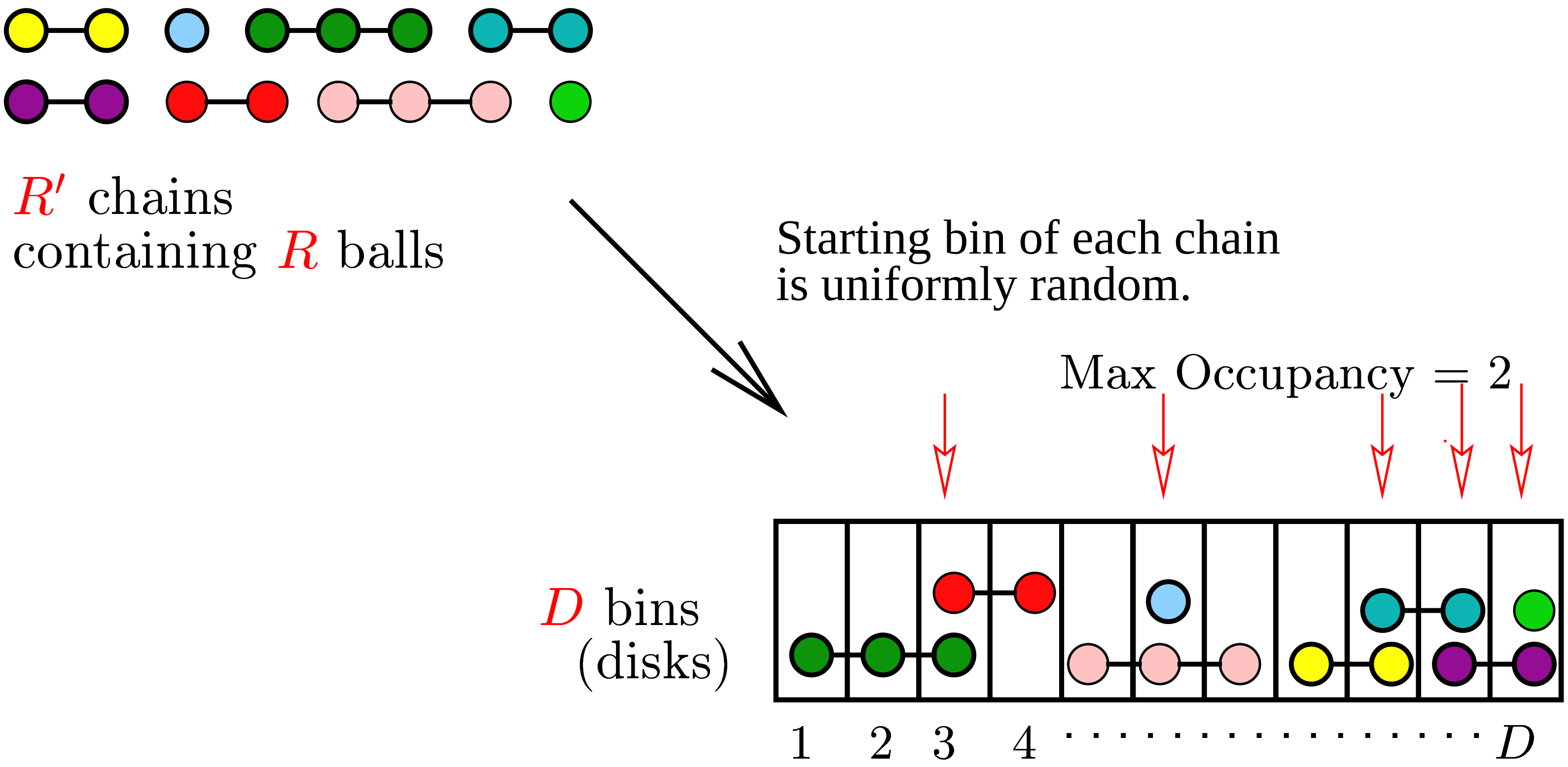
$$E[\#\mathrm{reads}_{SRM}] = E[\mathrm{MaxOcc}_{SRM}] \times \frac{n}{R} \times \lceil \log_R (n/m) \rceil$$

Classical Maximum Bucket Occupancy



$$E[\text{Classical Max Occupancy}] \sim \begin{cases} \dfrac{\ln D}{\ln\ln D} \cdot \dfrac{R}{D} & \text{if } \dfrac{R}{D} = 1 \\ c \cdot \dfrac{R}{D} & \text{if } \dfrac{R}{D} = \Theta(\log D) \\ \dfrac{R}{D} & \text{if } \dfrac{R}{D} \gg \log D \end{cases}$$

Occupancies with Dependency



R' chains
containing R balls

Starting bin of each chain
is uniformly random.

Max Occupancy $= 2$

D bins
(disks)

Conjecture: Let n_i =size of ith chain.

$$E[\mathrm{MaxOcc}_d(n_1, n_2, \ldots, n_{R'})]$$
$$\le \quad E[\mathrm{MaxOcc}_d(n_1, n_2, \ldots, n_{R'-1}, n_{R'} - 1, 1)]$$
$$\le \quad E[\text{Classical Max Occupancy}].$$

We can prove the (*harder !?!?*) conjecture if the balls of each chain are in random order rather than consecutive.

I/O Performance of SRM, $R = m/2$

Asymptotically,

$$E[\#\text{reads}_{SRM}] \leq \begin{cases} \dfrac{\ln D}{k \ln \ln D} \cdot \dfrac{n}{D} \log_m n & \text{if } \dfrac{m}{2D} \approx k. \\ c \cdot \dfrac{n}{D} \log_m n & \text{if } \dfrac{m}{2D} = \Theta(\log D). \\ \dfrac{n}{D} \log_m n & \text{if } \dfrac{m}{2D} \gg \log D. \end{cases}$$

Simulation of I/O performance ratio $\dfrac{\text{IO}_{\text{SRM}}}{\text{IO}_{\text{DSM}}}$ for $m \approx (2k+4)D$:

SRM is better than striping!

	$D = 5$	$D = 10$	$D = 50$
$k = 5$	0.56	0.47	0.37
$k = 10$	0.61	0.52	0.40
$k = 50$	0.71	0.63	0.51

Other Aspects

- ⋆ Probabilistic analysis required getting around dependencies.
- ⋆ Same technique and analysis for a simple randomized (multi-way) distribution. Application in parallel disk distribution sort.
- ⋆ Forecast and Flush technique has been used in a competitive parallel prefetching algorithm for certain request sequences, and may have other applications.
- ⋆ Not optimal theoretically for all parameter values because of maximum occupancy effect.

Implementation of SRM

- Implanting forecasting information.
- FF buffer management:
  - If # Free Blocks is $D - f$, Flush f “largest” blocks.
  - Forecast the “smallest” block from each disk.
  - Read in the “smallest” block from each disk.
- As stated, Forecasting requires D priority queues each containing R keys at any time.
- Flushing requires maintaining order among prefetched blocks in memory.

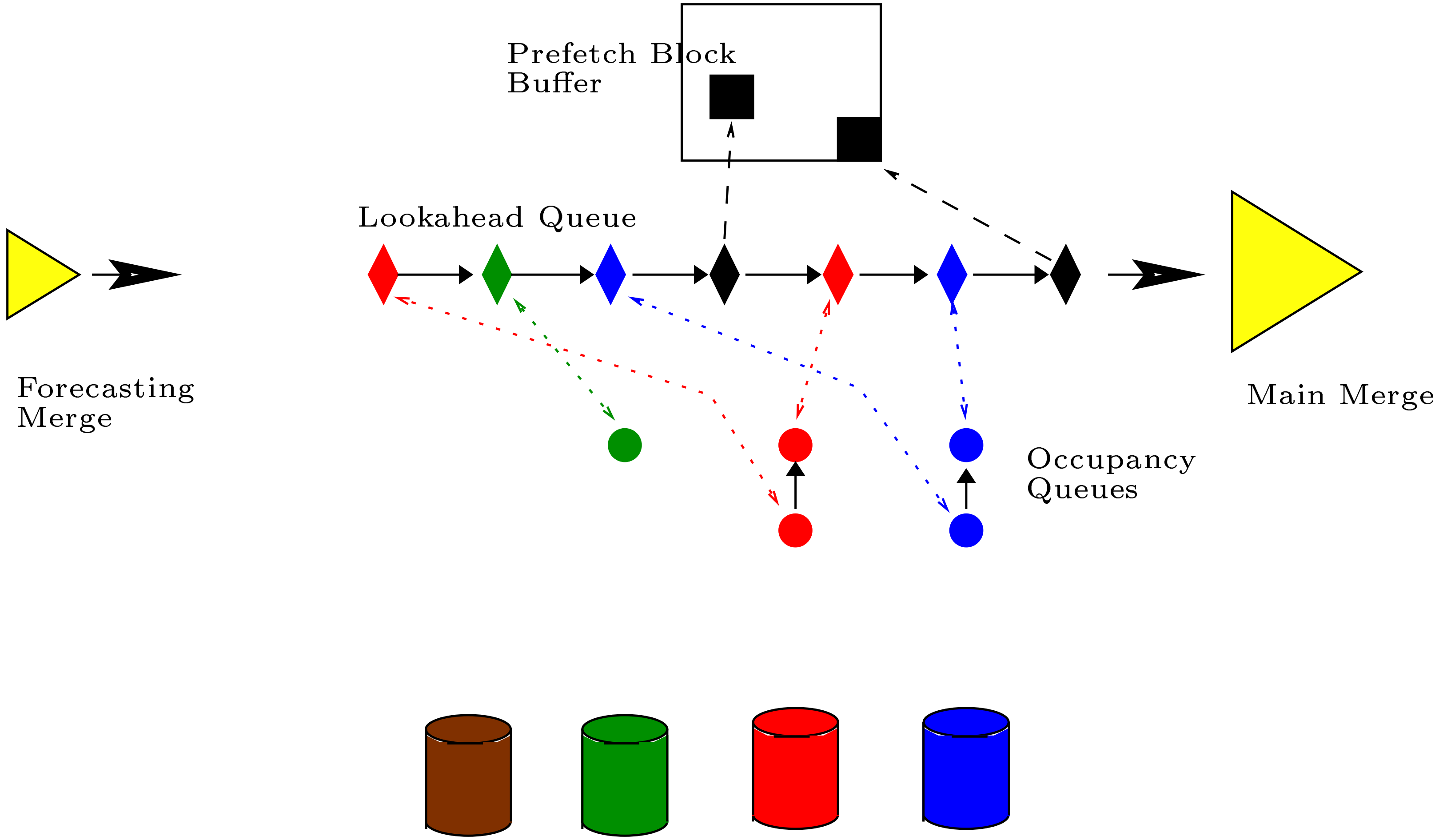
Simplifying the implementation of FF

- ☆ If I is item size,

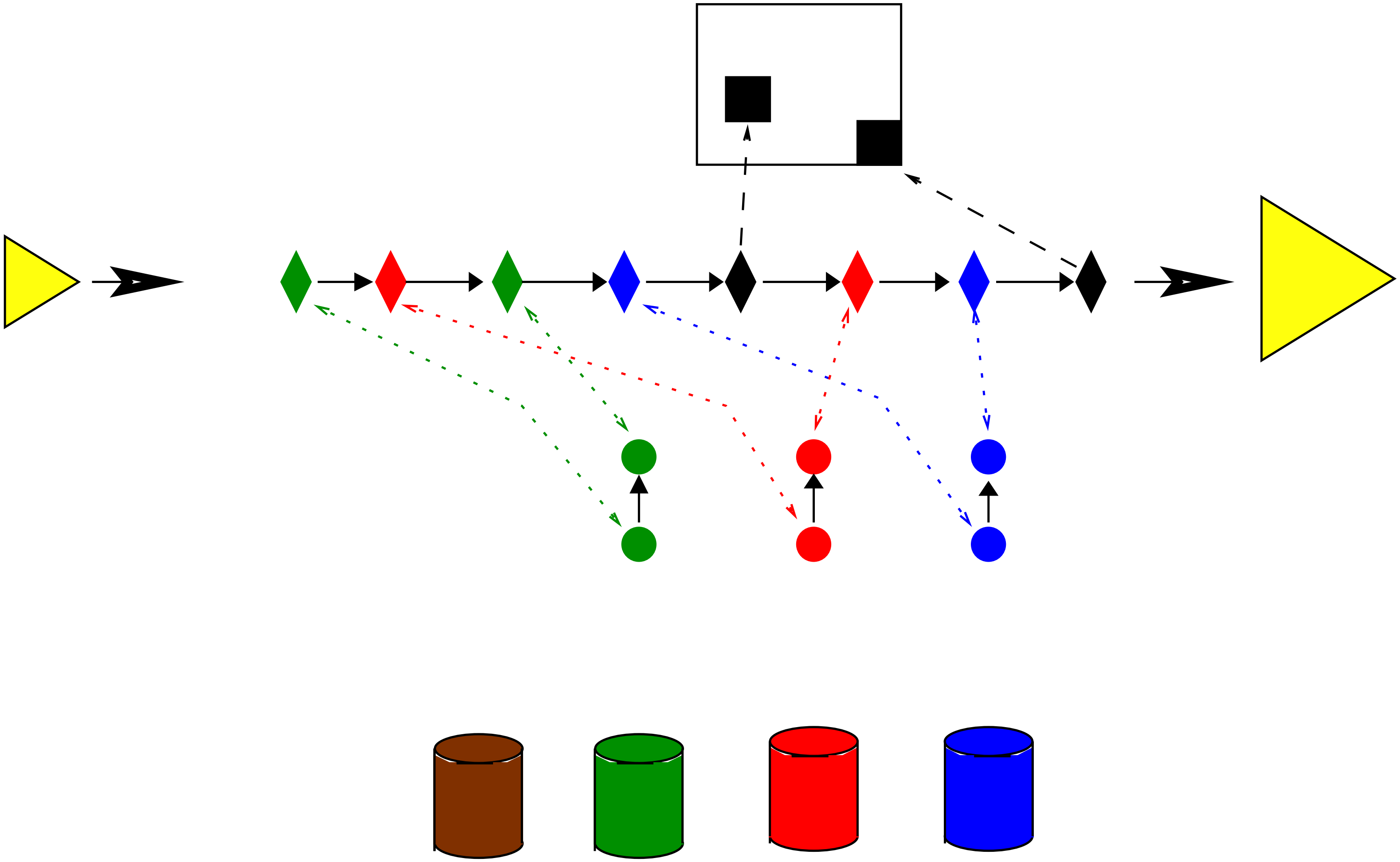
$$\text{Size of forecasting information} = \frac{1}{BI} \times \text{Input file size.}$$

- ☆ Our approach
  - *Don't implant* forecasting information in run blocks.
  - Store forecasting keys of a run in a separate file.
  - Using the forecasting keys during SRM requires a (much) smaller-scale R-way auxiliary merge of forecasting files.
- ☆ Requires only one priority queue with R keys in it.
- ☆ Automatically orders prefetched blocks and flushing is easy.

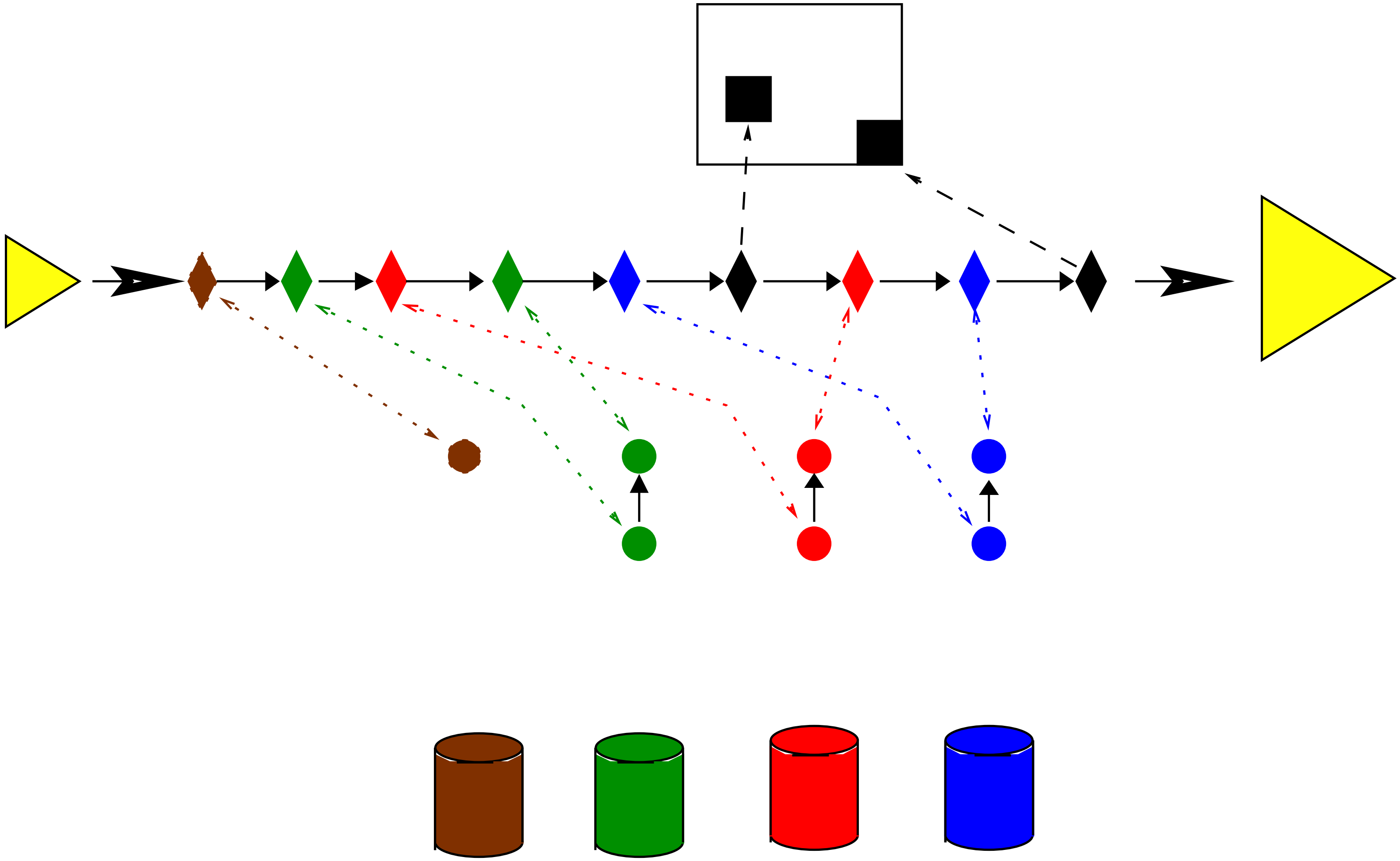
Simplified Implementation of FF



Simplified Implementation of FF

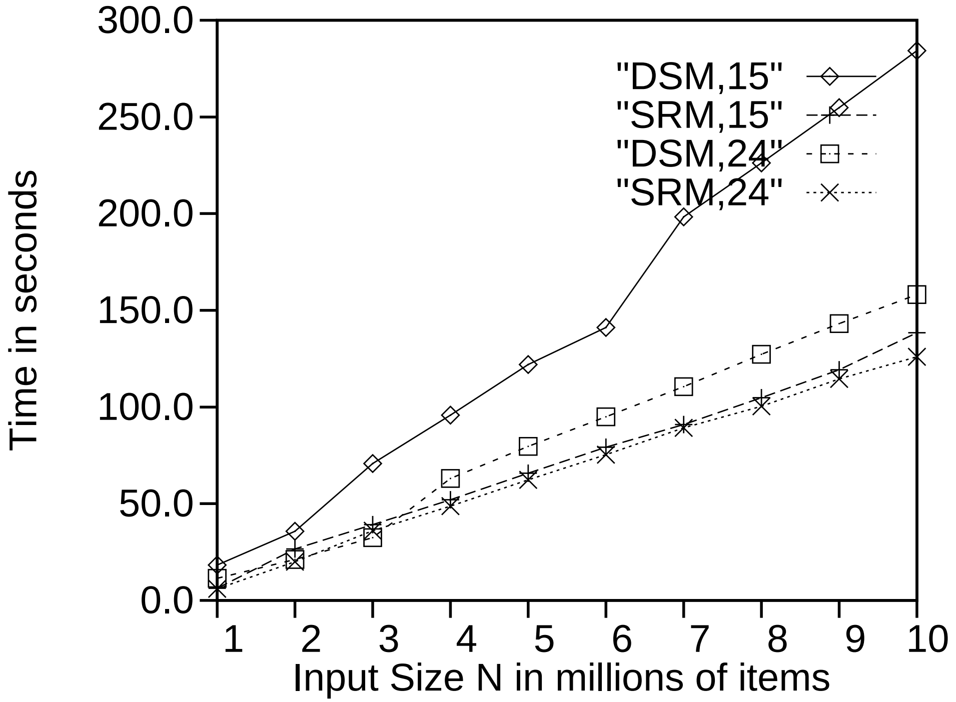


Simplified Implementation of FF



Practical Performance

- ☆ SRM outperforms DSM by 25–50% in running time.
- ☆ Merge passes of SRM are slower, but fewer than those of DSM.
- ☆ Overhead ratio for SRM is very close to 1 in practice.



Other Aspects

- Implementation of SRM and DSM was in the TPIE system; TPIE's functionality had to be extended to suport parallel disk operations.
- Several internal memory and other optimizations were programmed, and play a significant role in performance improvement.
- Parallel I/O was performed using the *mmb* memory-map system developed at Duke.

Conclusions

- ☆ Practical benefits from using parallel disks independently.
- ☆ SRM outperforms DSM by 25–50% in running time.
- ☆ Merge passes of SRM are slower, but fewer than those of DSM.
- ☆ Overhead ratio for SRM is very close to 1 in practice.
- ☆ Not theoretically optimal for all parameter settings (N, D, M, B).
- ☆ Stay tuned for distribution sort based on SRM ideas.
- ☆ Powerful notion of duality reconverts the distribution sort into merge sort that is provably optimal.