
Cache-Oblivious Algorithms

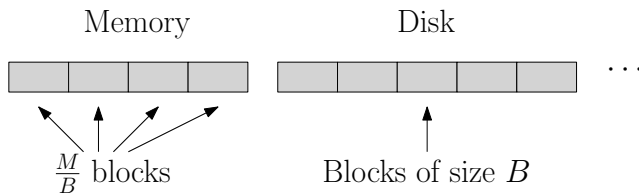
Matteo Frigo, Charles Leiserson, Harald Prokop, Sridhar
Ramchandran

Slides Written and Presented by William Kuszmaul

THE DISK ACCESS MODEL

Three Parameters:

B		Block Size in Words
M		Internal Memory Size in Words
P		Number of Concurrent Accesses Allowed (P is not considered in this paper)



Time is measured in *disk operations*.

FAST ALGORITHMS IN THE DISK ACCESS MODEL

$n \times n$ Matrix Multiplication: $O\left(\frac{n^3}{B\sqrt{M}}\right)$

Sorting: $O(n/B \cdot \log_M n)$

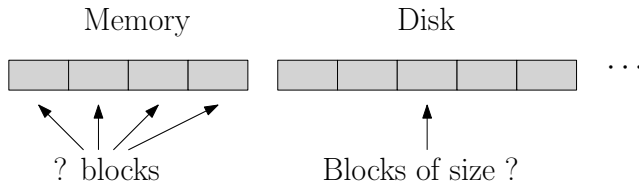
Fast Fourier Transform: $O(n/B \cdot \log_M n)$

(Running times given for $n \gg M \gg B$)

THIS PAPER: CACHE-OBLIVIOUS ALGORITHMS

The Setup:

- ▶ Algorithm *oblivious* to M and B
- ▶ Still evaluated in Disk Access Model



Question: Can we still get good running times?

WHY CACHE-OBLIVIOUS ALGORITHMS?

Advantages:

- ▶ Don't need to be tuned to specific machine
- ▶ Can interact well with *multiple caches* concurrently
- ▶ Algorithmically cool

Disadvantages:

- ▶ Are they practical? (Actually they often are!)

ALGORITHMS IN THIS PAPER

$n \times n$ Matrix Multiplication: $O\left(\frac{n^3}{B\sqrt{M}}\right)$

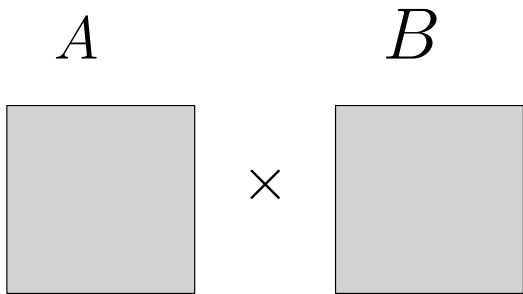
Sorting: $O(n/B \cdot \log_M n)$

Fast Fourier Transform: $O(n/B \cdot \log_M n)$

(Running times given for $n \gg M \gg B$)

Part 1: Matrix Multiplication

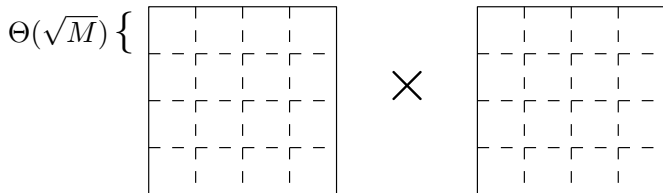
THE SETUP: MULTIPLYING TWO $n \times n$ MATRICES



Simplifying Assumptions:

- ▶ $n \gg M \gg B$
- ▶ n is a power of two

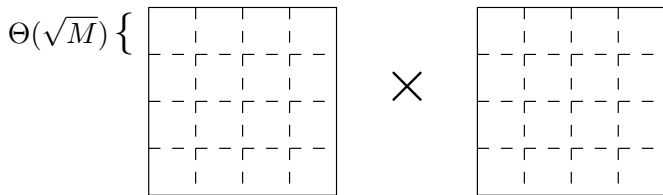
NON-OBLIVIOUS TILING ALGORITHM



The Algorithm:

- ▶ **Step 1:** Break matrices into tiles of size $\Theta(\sqrt{M})$
- ▶ **Step 2:** Treat each tile as a “number” and do normal matrix multiplication

NON-OBLIVIOUS TILING ALGORITHM

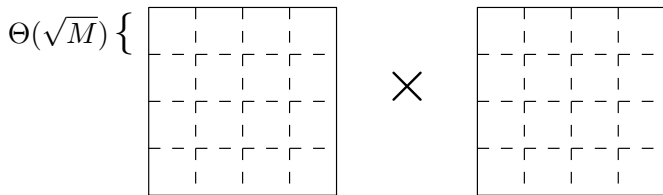


Running Time:

- ▶ Multiplying two tiles takes time:

$O(M/B)$ instead of $O(\sqrt{M}^3)$.

NON-OBLIVIOUS TILING ALGORITHM



Running Time:

- ▶ Multiplying two tiles takes time:

$$O(M/B) \text{ instead of } O(\sqrt{M}^3).$$

- ▶ Total running time:

$$O\left(\frac{n^3}{B\sqrt{M}}\right).$$

CACHE-OBLIVIOUS MATRIX MULTIPLICATION

$$\begin{array}{c} A \\ \begin{array}{|c|c|} \hline A_1 & A_2 \\ \hline A_3 & A_4 \\ \hline \end{array} \end{array} \times \begin{array}{c} B \\ \begin{array}{|c|c|} \hline B_1 & B_2 \\ \hline B_3 & B_4 \\ \hline \end{array} \end{array}$$

The Algorithm:

- ▶ **Step 1:** Tile each matrix into fourths
- ▶ **Step 2:** Treat each tile as a “number” and multiply the 2×2 matrices.
- ▶ **Recursion:** When multiplying each A_i and B_j , recursively repeat entire procedure.

CACHE-OBLIVIOUS MATRIX MULTIPLICATION

$$\begin{array}{c} A \\ \begin{array}{|c|c|} \hline A_1 & A_2 \\ \hline A_3 & A_4 \\ \hline \end{array} \end{array} \times \begin{array}{c} B \\ \begin{array}{|c|c|} \hline B_1 & B_2 \\ \hline B_3 & B_4 \\ \hline \end{array} \end{array}$$

Running Time:

- ▶ **Simulates Standard Tiling:** Once recursive tile-size becomes $\leq M$, the multiplications will be done in memory
- ▶ Total running time:

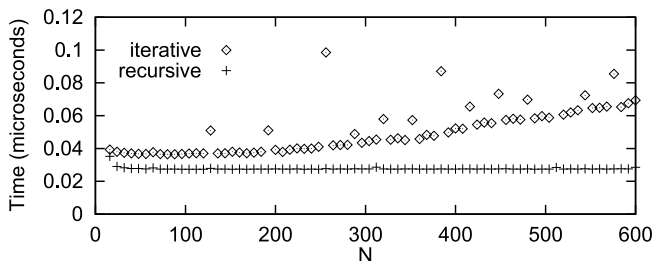
$$O\left(\frac{n^3}{B\sqrt{M}}\right).$$

HANDLING NON-SQUARE MATRICES

$$\begin{array}{|c|c|} \hline & A \\ \hline A_1 & \vdots & A_2 \\ \hline \end{array} \times \begin{array}{|c|} \hline B \\ \hline B_1 \\ \hline \text{---} \\ \hline B_2 \\ \hline \end{array}$$

Key Idea: Split long direction in two and recurse.

REAL-WORLD COMPARISON TO NAIVE n^3 ALGORITHM



- Average time taken to multiply two $N \times N$ matrices, divided by N^3 .
 - ▶ How does this compare to tiled algorithm? They don't say.

WHY DO WE NEED $M \gg B$?

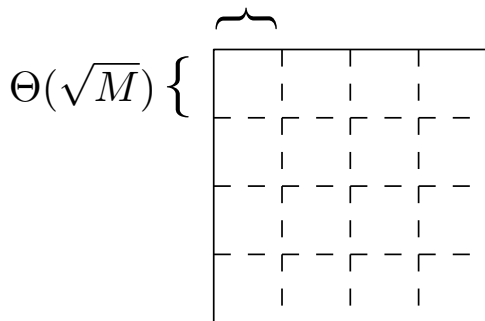
- ▶ Tiling algorithms require $M \geq B^2$.
- ▶ Known as the *tall cache assumption* because means:
Number of blocks in cache \geq Size of each block

WHY DO WE NEED $M \gg B$?

- ▶ Tiling algorithms require $M \geq B^2$.
- ▶ Known as the *tall cache assumption* because means:
Number of blocks in cache \geq Size of each block

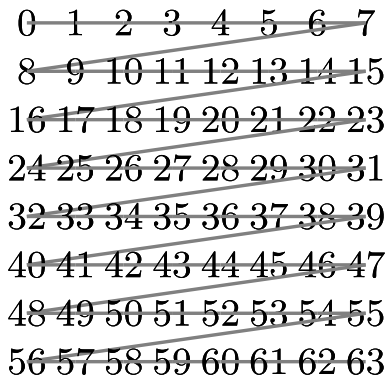
Why we need it:

Need this to be $\Omega(B)$



ELIMINATING THE TALL CACHE ASSUMPTION

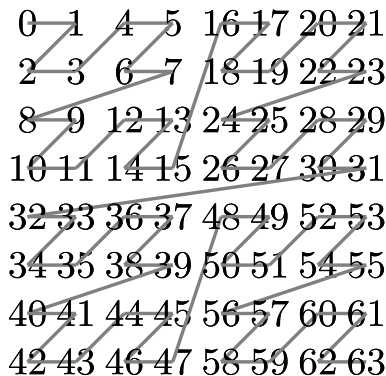
The Key Idea: Change how we store matrices!



A diagram showing a matrix with 8 rows and 8 columns. The numbers 0 through 63 are arranged in a standard row-major order. Each row is crossed out with a single horizontal line, representing a single row being accessed sequentially.

0 1 2 3 4 5 6 7
8 9 10 11 12 13 14 15
16 17 18 19 20 21 22 23
24 25 26 27 28 29 30 31
32 33 34 35 36 37 38 39
40 41 42 43 44 45 46 47
48 49 50 51 52 53 54 55
56 57 58 59 60 61 62 63

Normal Ordering



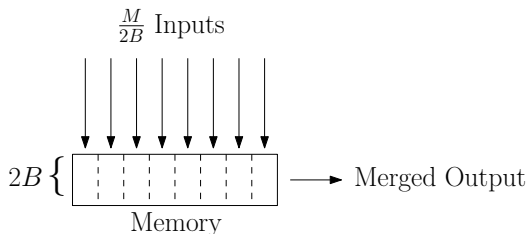
A diagram showing a matrix with 8 rows and 8 columns. The numbers 0 through 63 are arranged in a cache-oblivious order. The numbers in each row are 0-7, 2-7, 8-13, 10-15, 32-37, 34-39, 40-45, and 42-47. Lines connect the numbers in each row to their corresponding positions in the normal ordering matrix, showing that the cache-oblivious order interleaves elements from different rows of the normal matrix.

0 1 4 5 16 17 20 21
2 3 6 7 18 19 22 23
8 9 12 13 24 25 28 29
10 11 14 15 26 27 30 31
32 33 36 37 48 49 52 53
34 35 38 39 50 51 54 55
40 41 44 45 56 57 60 61
42 43 46 47 58 59 62 63

Cache-Oblivious Ordering

Part 2: Sorting

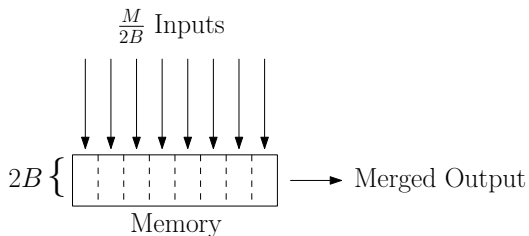
MERGESORT IN THE DISK ACCESS MODEL



Key Idea: Performing $\frac{M}{2B}$ -way merges

- ▶ Assign to each input stream a buffer of size $2B$
- ▶ Read a block from input stream when buffer \leq half full
- ▶ At each step output the B smallest elements in buffers

MERGESORT IN THE DISK ACCESS MODEL



Running Time:

- ▶ $O(\log_{M/B} n)$ levels of recursion
- ▶ Each takes time $O(n/B)$
- ▶ **Total Running Time:** $O\left(\frac{n}{B} \log_M n\right)$

(Assuming $n \gg M \gg B$)

CACHE-OBLIVIOUS SORTING

This paper introduces two algorithms:

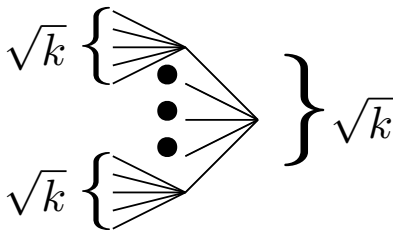
Funnel Sort: A cache-oblivious merge sort
(We will focus on this one)

Modified Distribution Sort: Based on another
Disk-Access-Model Algorithm.

A FAILED ATTEMPT AT CACHE-OBLIVIOUS MERGING

Question: How to we merge k streams?

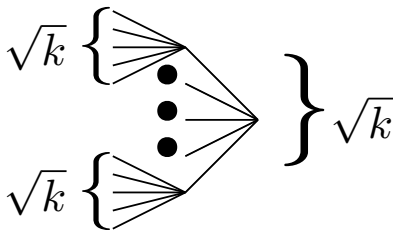
Answer: Recursively with \sqrt{k} -merges:



A FAILED ATTEMPT AT CACHE-OBLIVIOUS MERGING

Question: How to we merge k streams?

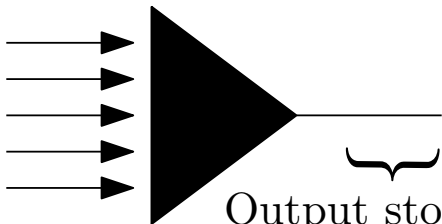
Answer: Recursively with \sqrt{k} -merges:



Wait a second... This reduces to normal merge sort!

k -MERGERS IN FUNNEL SORT

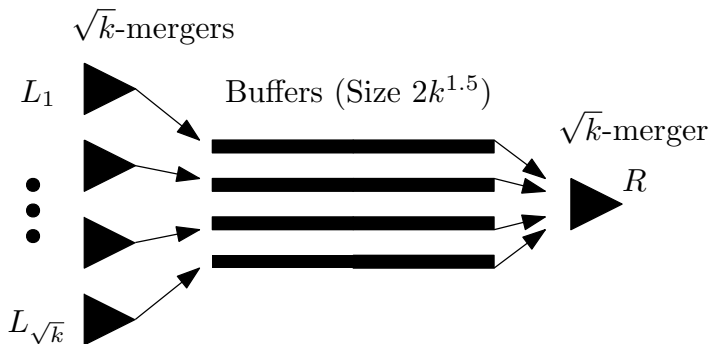
k streams



Output stops
after k^3 elts

- ▶ Merges k input streams
- ▶ **Critical Caveat:** Each invocation of k -merger only outputs k^3 elements
- ▶ Full k -merge may require multiple invocations!

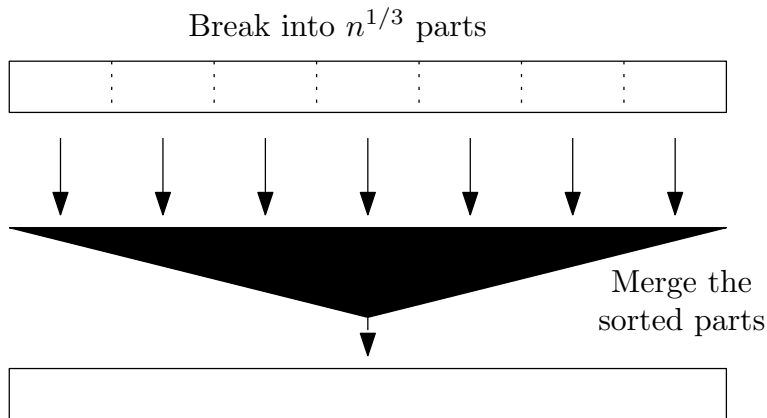
RECURSIVE k -MERGERS



Building k -merger out of \sqrt{k} -Mergers:

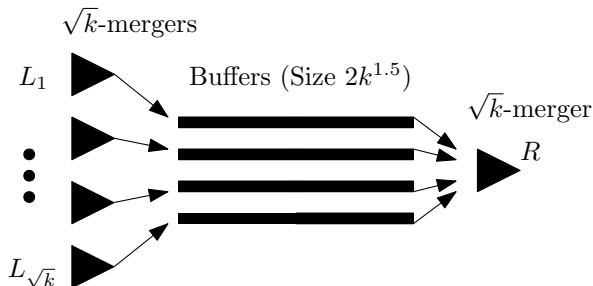
- ▶ Need to invoke R a total of $k^{1.5}$ times
- ▶ Before each invocation of R :
 - ▶ Check if any buffers less than half full
 - ▶ Invoke L_i 's to refill such buffers

SORTING WITH k -MERGERS



- ▶ **Step 1:** Break array into $n^{1/3}$ sub-arrays of size $n^{2/3}$
- ▶ **Step 2:** Recursively sort each sub-array
- ▶ **Step 3:** Perform a $n^{1/3}$ -merger on the sub-arrays

HOW MUCH WORK IN RAM MODEL?

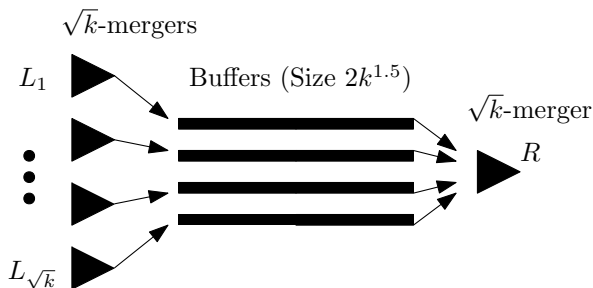


Key Insight: Essentially just merge sort with merges interleaved strangely.

Running Time in RAM Model: $O(n \log n)$

But What About in the Disk Access Model?

KEY PROPERTY OF k -MERGERS



Key Property: Each invocation of a k -merger has memory footprint $O(k^3)$.

Consequence: $M^{1/3}$ -mergers can be performed in memory.

RUNNING TIME IN DISK ACCESS MODEL

In RAM model, each $M^{1/3}$ -merger takes time:

$$\Theta(M \cdot \log M).$$

In Disk Access Model, each $M^{1/3}$ -merger takes time:

$$\Theta(M/B).$$

Full sorting time in disk access model:

$$\Theta\left(\frac{n \log n}{B \log M}\right) = \Theta\left(\frac{n}{B} \cdot \log_M n\right).$$

(Assuming $n \gg M \gg B$ and ignoring some details)

IS FUNNEL SORT PRACTICAL?

See the next talk!