Cache-Oblivious Algorithms

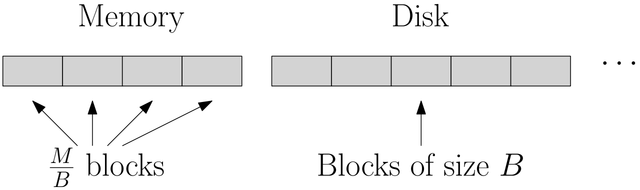
Matteo Frigo, Charles Leiserson, Harald Prokop, Sridhar Ramchandran

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

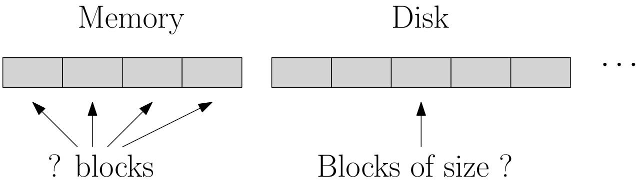
Fast Fourier Transform: $O(n/B \cdot \log_M n)$

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

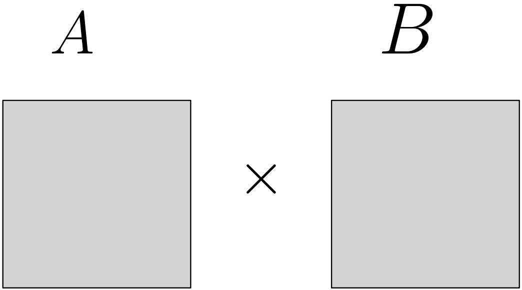
Sorting: $O(n/B \cdot \log_M n)$

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

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

Part 1: Matrix Multiplication

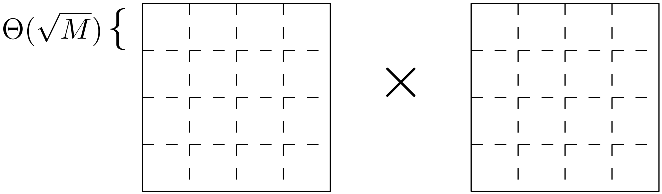
THE SETUP: MULTIPLYING TWO $n \times n$ MATRICES



Simplifying Assumptions:

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

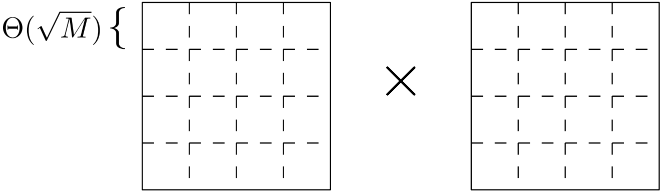
Non-Oblivious Tiling Algorithm



The Algorithm:

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

Non-Oblivious Tiling Algorithm

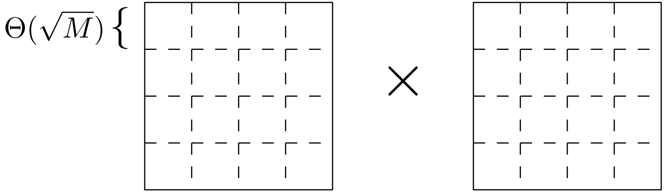


Running Time:

- Multiplying two tiles takes time:

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

Non-Oblivious Tiling Algorithm



Running Time:

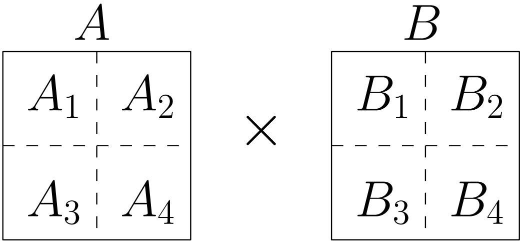
- Multiplying two tiles takes time:

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

- Total running time:

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

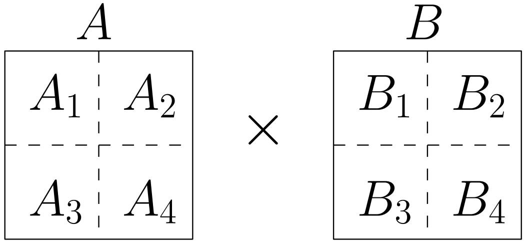
Cache-Oblivious Matrix Multiplication



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

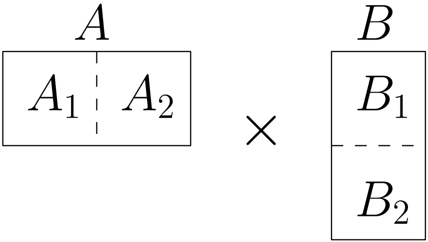


Running Time:

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

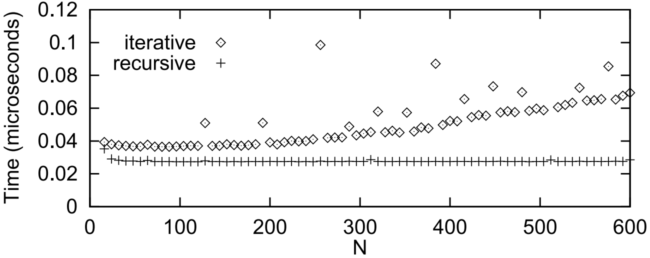
$$O\left(\frac{n^3}{B\sqrt{M}}\right).$$

Handling Non-Square Matrices



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 $M \geqslant B^2$.
- Known as the *tall cache assumption* because means:

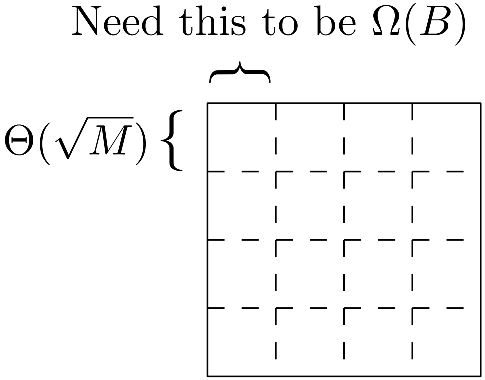
  Number of blocks in cache $\geqslant$ Size of each block

WHY DO WE NEED $M \gg B$?

- Tiling algorithms require $M \geqslant B^2$.
- Known as the *tall cache assumption* because means:

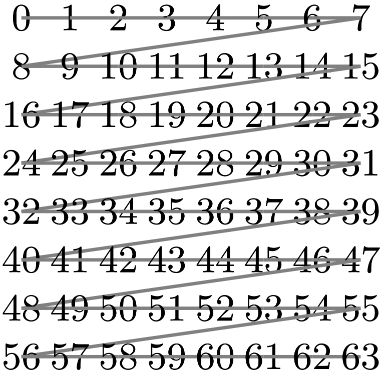
  Number of blocks in cache $\geqslant$ Size of each block

Why we need it:

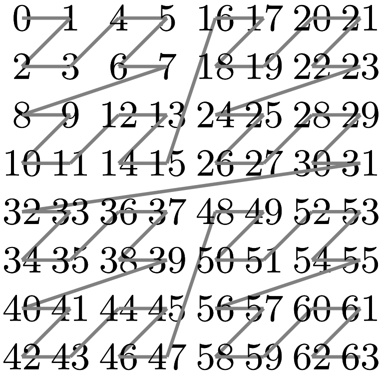


ELIMINATING THE TALL CACHE ASSUMPTION

The Key Idea: Change how we store matrices!



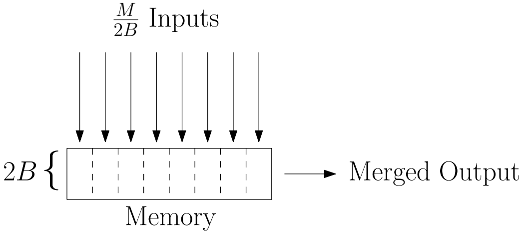
Normal Ordering



Cache-Oblivious Ordering

Part 2: Sorting

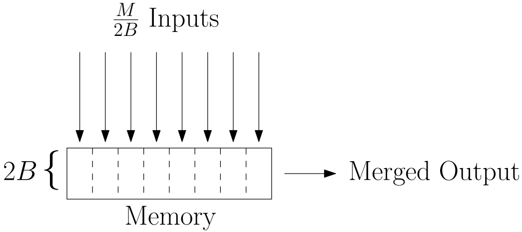
Mergesort in the Disk Access Model



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 $\leqslant$ 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:

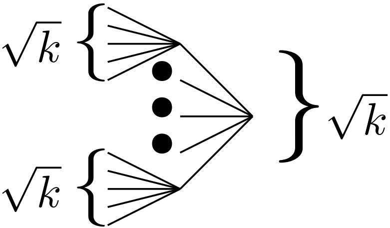
Funnel Sort: A cache-oblivious merge sort
(We will focus on this one)

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

A Failed Attempt at Cache-Oblivious Merging

Question: How to we merge k streams?

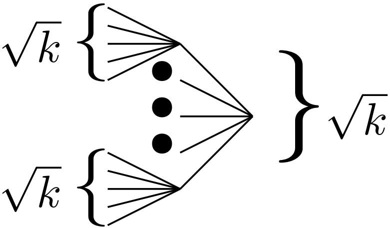
Answer: Recursively with $\sqrt{k}$-merges:



A Failed Attempt at Cache-Oblivious Merging

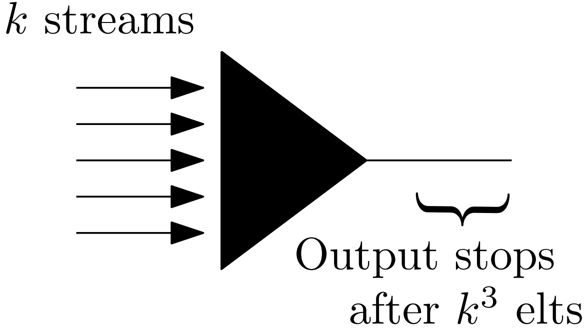
Question: How to we merge k streams?

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



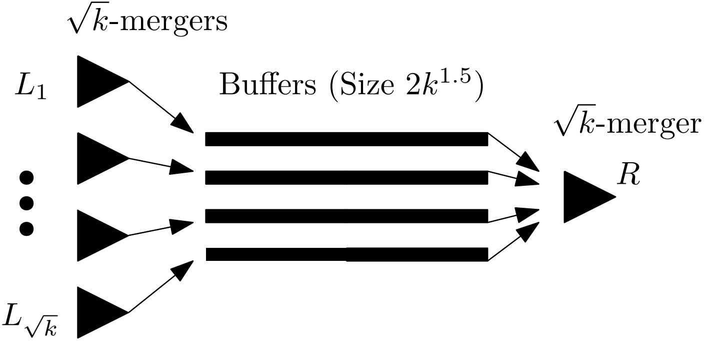
Wait a second... This reduces to normal merge sort!

k-Mergers in Funnel Sort



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

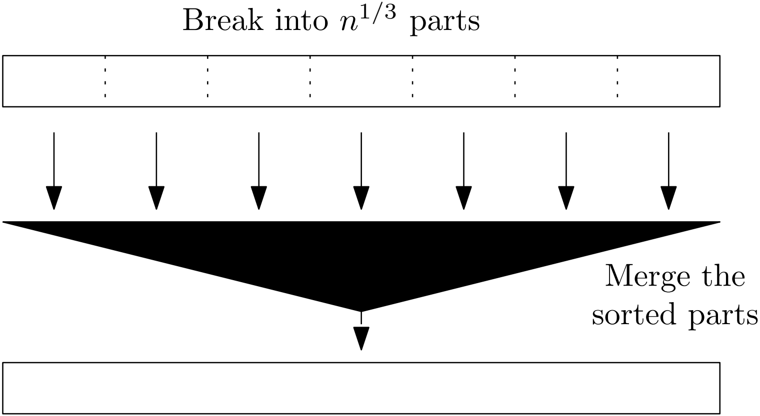
RECURSIVE k-MERGERS



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

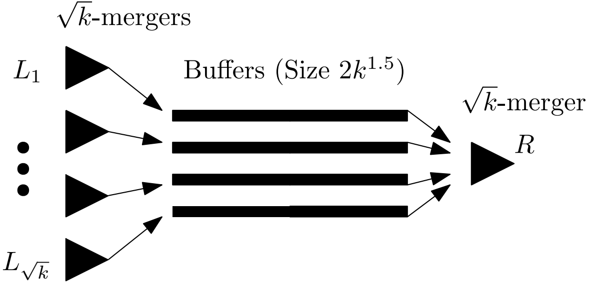
- Need to invoke R a total of $k^{1.5}$ times
- Before each invocation of R:
  - Check if any buffers less than half full
  - Invoke L_i's to refill such buffers

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?

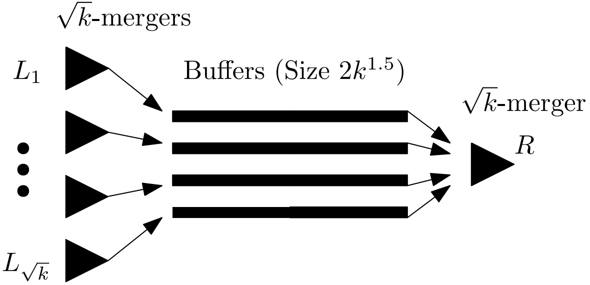


Key Insight: Essentially just merge sort with merges interleaved strangely.

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!