Cache-Oblivious Algorithms

Matteo Frigo, Charles Leiserson, Harald Prokop, Sridhar Ramachandran

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)

Memory

\[ \frac{M}{B} \text{ blocks} \]

Disk

Blocks of size $B$

Time is measured in *disk operations*. 
Fast Algorithms in the Disk Access Model

\[ \text{Matrix Multiplication: } O \left( \frac{n^3}{B\sqrt{M}} \right) \]

\[ \text{Sorting: } O(n/B \cdot \log_M n) \]

\[ \text{Fast Fourier Transform: } O(n/B \cdot \log_M n) \]

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

\[ \Theta(\sqrt{M}) \{ \begin{array}{cccc} \_ & \_ & \_ & \_ \\ \_ & \_ & \_ & \_ \\ \_ & \_ & \_ & \_ \\ \_ & \_ & \_ & \_ \end{array} \times \begin{array}{cccc} \_ & \_ & \_ & \_ \\ \_ & \_ & \_ & \_ \\ \_ & \_ & \_ & \_ \\ \_ & \_ & \_ & \_ \end{array} \] 

Running Time:
- Multiplying two tiles takes time:

\[ O(M/B) \] instead of \[ O(\sqrt{M^3}) \].
**Non-Oblivious Tiling Algorithm**

\[ \Theta(\sqrt{M}) \{ \begin{array}{cc} \times & \end{array} \} \]

**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 $2 \times 2$ matrices.
- **Recursion:** When multiplying each $A_i$ and $B_j$, recursively repeat entire procedure.
**Cache-Oblivious Matrix Multiplication**

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

**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

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:**

\[
\Theta(\sqrt{M}) \left\{ \begin{array}{c}
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\vdots 
\end{array} \right.
\]

\[\text{Need this to be } \Omega(B)\]
**Eliminating the Tall Cache Assumption**

**The Key Idea:** Change how we store matrices!

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

![Diagram of Mergesort](image)

**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 \(n \gg M \gg B\))
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 do we merge $k$ streams?

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

$$\sqrt{k} \left\{ \begin{array}{c} \cdot \\ \cdot \\ \cdot \end{array} \right\} \sqrt{k}$$

Wait a second... This reduces to normal merge sort!
A FAILED ATTEMPT AT CACHE-OBLIVIOUS MERGING

**Question:** How to we merge $k$ streams?

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

\[
\sqrt{k} \quad \{ \quad \bullet \quad \bullet \quad \bullet \\
\quad \} \quad \sqrt{k}
\]

Wait a second... This reduces to normal merge sort!
**$k$-Mergers in Funnel Sort**

- **$k$ 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

Break into $n^{1/3}$ parts

- **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**

$\sqrt{k}$-mergers

$L_1$ Buffers (Size $2k^{1.5}$)

$L_{\sqrt{k}} \sqrt{k}$-merger

**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:

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

(Assuming $n \gg M \gg B$ and ignoring some details)
Is funnel sort practical?

See the next talk!