# Lecture 23: Cache-oblivious Algorithms I

This lecture introduces cache-oblivious algorithms. Topics include

- memory hierarchy
- external memory vs. cache-oblivious model
- cache-oblivious scanning
- cache-oblivious divide & conquer algorithms: median finding & matrix multiplication

## 1 Modern Memory Hierarchy

So far in this class, we have viewed all operations and memory accesses as equal cost. However, modern computers have memory hierarchy.

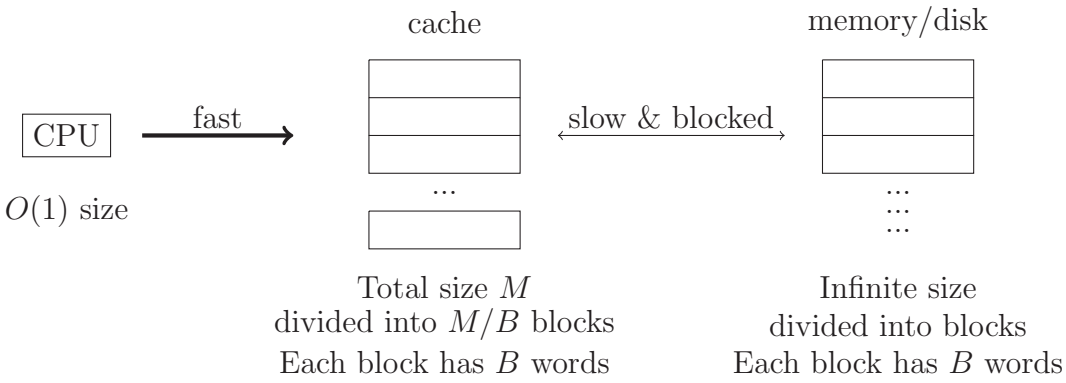
| CPU | ⟷ | L1 | ⟷ L2 | ⟷ L3 | ⟷ memory | ⟷ disk |
|---|---|---|---|---|---|---|
| size | ∼ | 10KB | 100KB | MB | GB | TB |
| latency | ∼ | 1ns | 10ns | 10ns | 100ns | 10ms |

Each hierarchy on the right is bigger, but has longer latency due to the longer distance data has to travel. Yet, bandwidth between different hierarchies is usually matched.

A common technique to mitigate latency is *blocking*: when fetching a word of data, get the entire block containing it. Using algorithmic terminology, the idea is to amortize latency over a whole block. For this idea to work, we additional require the algorithm to use all words in a block (spatial locality) and reuse blocks in cache (temporal locality).

# 2 Memory Model of Algorithms

## 2.1 External Memory Model




In this model, cache acecsses are free. The algorithm explicitly reads and writes memory in blocks. We will count the number of memory transfers between the cache and the memory, as an important metric of the algorithm's performance.

## 2.2 Cache-oblivious Model

The only change from the external memory model is that the algorithm no longer knows $B$ and $M$. Accessing a word in the memory automatically fetches an entire block into the cache, and evicts the least recently used (LRU) block from the cache if the cache is full.

Every algorithm is a cache-oblivious algorithm, but we would like to find the algorithm that minimizes our new metirc — the number of memory transfers.

Why do we like cache-olivious algorithms (as opposed to letting the algorithm know $B$ and $M$)? Because this way, an algorithm can auto-tune itself and run efficiently on different computers (possibly with different $B$ and $M$). Besides, it is cool research!

# 3 Scanning

Example program:

for $i$ in range($N$): sum += $A[i]$

Assume $A$ is stored contiguously in memory. External memory model can align $A$ with a block boundary, so it needs $\lceil N/B \rceil$ memory transfers.

Cache-oblivious algorithms cannot control alignment (because it does not know $B$), so it needs $\lceil N/B \rceil + 1 = N/B + O(1)$ memory transfers. $O(1)$ parallel scans still need $O(N/B + 1)$ memory transfers.

# 4 Divide & Conquer

Divide & Conquer algorithms divide problems down to $O(1)$ size. The base case of the recursion is either when

- problem fits in cache i.e., $\leq M$, or
- problem fits in $O(1)$ blocks, i.e., $O(B)$.

Below we will see one example for each.

## 4.1 Median Finding / Order Statistics

Recall the steps of the algorithm

1. view array as partitioned into columns of 5 (each column is $O(1)$ size).
2. sort each column
3. recursively find the median of the column medians
4. partition array by $x$
5. recurse on one side

We will now analyze the number of memory transfers in each step. Let $MT(N)$ be the total number of memory transfers.

1. free
2. a scan, $O(N/B + 1)$
3. $MT(N/5)$, this involves a pre-processing step that coallesces the $N/5$ elements in a consecutive array
4. 3 parallel scans, $O(N/B + 1)$
5. $MT(7N/10)$

Therefore, we get the recursion

$$MT(N) = MT(N/5) + MT(7N/10) + O(N/B + 1).$$

Solving the recursion requires setting a base case. An obvious base case is $MT(O(1)) = O(1)$.

But we can get a stronger base case: $MT(O(B)) = O(1)$. Uing this base case, the recursion solves to $MT(N) = O(N/B + 1)$. (Intuition: cost at level of the recursion decreases geometrically, so the cost at root dominates.)

## 4.2 Matrix Multiplication

Problem: compute $Z = X \cdot Y$ where $X, Y, Z$ are all $N \times N$ matrices. Also suppose $X$ is stored in row-major order, and $Y$ is stored in column-major order to improve locality.

If we use the basic algorithm, computing one element in $Z$ requires one or two parallel scans, because it either requires scanning either a new row from $X$, or a new column from $Y$. Computing each element takes $O(N/B + 1)$ memory transfers, so computing the entire $Z$ costs $O(N^3/B + N^2)$ memory transfers.

Instead we will use the blocked matrix multiplcation algorithm (not Strassen). Note that key block should be stored consecutively. We get recursion

$$MT(N) = 8MT(N/2) + O(N^2/B + 1)$$

The first term is recursive sub-matrix multiplication, and the second term is matrix addition which requires scanning the matrices.

Again, we can have different base cases

- weak: $MT(O(1)) = O(1)$
- better: $MT(O(B)) = O(1)$
- even better: $MT(\sqrt{M/3}) = O(M/B)$

The third case represents the case that the three involved matrices can fit in the cache together. Therefore, to multiply them, we only need one scan to load all of them into cache.

If we draw the recursion tree, the cost at each level is geometrically increasing this time, $N^2/B, 8(\frac{N}{2})^2/B, 8^2(\frac{N}{4})^2/B, \ldots$. Therefore, the cost at the leaves dominate, and the total cost = cost per leave · number of leaves,

$$MT(N) = O(M/B) \cdot 8^{O(\lg N/\sqrt{M})} = O(M/B) \cdot O((N/\sqrt{M})^3) = O(N^3/B\sqrt{M}).$$

6.046J / 18.410J Design and Analysis of Algorithms
Spring 2015