15-853: Algorithms in the Real World

Locality II: Cache-oblivious algorithms
- Matrix multiplication
- Distribution sort
- Static searching

I/O Model
Abstracts a single level of the memory hierarchy
- Fast memory (cache) of size $M$
- Accessing fast memory is free, but moving data from slow memory is expensive
- Memory is grouped into size-$B$ blocks of contiguous data
- Cost: the number of block transfers (or I/Os) from slow memory to fast memory.

Cache-Oblivious Algorithms
- Algorithms not parameterized by $B$ or $M$.
  - These algorithms are unaware of the parameters of the memory hierarchy
- Analyze in the ideal cache model — same as the I/O model except optimal replacement is assumed
- Optimal replacement means proofs may posit an arbitrary replacement policy, even defining an algorithm for selecting which blocks to load/evict.

Advantages of Cache-Oblivious Algorithms
- Since CO algorithms do not depend on memory parameters, bounds generalize to multilevel hierarchies.
- Algorithms are platform independent
- Algorithms should be effective even when $B$ and $M$ are not static
Matrix Multiplication

Consider standard iterative matrix-multiplication algorithm.

\[
\begin{align*}
Z & := X \cdot Y \\
\text{for } i = 1 \text{ to } N \text{ do} \\
& \quad \text{for } j = 1 \text{ to } N \text{ do} \\
& \quad \text{for } k = 1 \text{ to } N \text{ do} \\
& \quad \quad Z[i][j] := X[i][k] \cdot Y[k][j] \\
\end{align*}
\]

- Where \(X, Y,\) and \(Z\) are \(N \times N\) matrices.
- \(\Theta(N^3)\) computation in RAM model. What about I/O?

How Are Matrices Stored?

How data is arranged in memory affects I/O performance.

- Suppose \(X, Y,\) and \(Z\) are in row-major order.

\[
\begin{align*}
Z & := X \cdot Y \\
\text{for } i = 1 \text{ to } N \text{ do} \\
& \quad \text{for } j = 1 \text{ to } N \text{ do} \\
& \quad \text{for } k = 1 \text{ to } N \text{ do} \\
& \quad \quad Z[i][j] := X[i][k] \cdot Y[k][j] \\
\end{align*}
\]

- If \(N \geq B\), reading a column of \(Y\) is expensive = \(\Theta(N)\) I/Os.
- If \(N \geq M\), no locality across iterations for \(X\) and \(Y\) = \(\Theta(N^3)\) I/Os.

We can do much better than \(\Theta(N^3/B)\) I/Os, even if all matrices are row-major.

Recursive Matrix Multiplication

Summing two matrices with row-major layout is cheap — just scan the matrices in memory order.

- Cost is \(\Theta(N^2/B)\) I/Os to sum two \(N \times N\) matrices, assuming \(N \geq B\).
Recursive Multiplication Analysis

The big question is the base case:
- Suppose an $X$, $Y$, and $Z$ submatrices fit in memory at the same time
- Then multiplying them in memory is free after paying $\Theta(M/B)$ to load them into memory

How Big a Matrix Fits in Memory?

Does a $c\sqrt{M} \times c\sqrt{M}$ submatrix fit in memory?
- Let's consider each row:
  - If $\sqrt{M} \geq B$, then each row occupies $\Theta(\sqrt{M}/B)$ blocks.
  
- $c\sqrt{M}$ rows use $\Theta(M/B)$ blocks

Recursive Multiplication Analysis

Assuming $M \geq B^2$:
- $\text{Mult}(cM) = \Theta(M/B)$ to load small matrices into memory
- $\text{Mult}(N) = 8\text{Mult}(N/2) + O(N^2/B)$
- Solves to $\text{Mult}(N) = \Theta(N^3/BM + N^2/B)$
Without Tall-Cache Assumption

Try a better matrix layout
• The algorithm is recursive. Use a layout that matches the recursive nature of the algorithm
• For example, Z-morton ordering:
  - The line connects elements that are adjacent in memory
  - In other words, construct the layout by storing each quadrant of the matrix contiguously, and recurse

Recursive MatMul with Z-Morton

The analysis becomes easier
• Each quadrant of the matrix is contiguous in memory, so a $cM \times cM$ submatrix fits in memory
  - The tall-cache assumption is not required to make this base case work
  - The rest of the analysis is the same

Searching: binary search is bad

Example: binary search for element A with block size $B = 2$
• Search hits a different block until reducing keyspace to size $\Theta(B)$.
• Thus, total cost is $\log_2 N = \Theta(\log_2 B) = \Theta(\log_2 (N/B)) \approx \Theta(\log_2 N)$ for $N \gg B$

Static cache-oblivious searching

Goal: organize $N$ keys in memory to facilitate efficient searching. (van Emde Boas layout)
1. build a balanced binary tree on the keys
2. layout the tree recursively in memory, splitting the tree at half the height
Static layout example

Cache-oblivious searching: Analysis I
- Consider recursive subtrees of size $\sqrt{B}$ to $B$ on a root-to-leaf search path.
- Each subtree is contiguous and fits in $O(1)$ blocks.
- Each subtree has height $\Theta(\log B)$, so there are $\Theta(\log B)$ of them.

Cache-oblivious searching: Analysis II
- $S(N) = 2S(\sqrt{N})$ or $S(N) = 2S(\sqrt{N}) + O(1)$
- Base case $S(\leq B) = 0$
- Solve to $O(\log_B N)$

Distribution sort outline
- Analogous to multiway quicksort
- 1. Split input array into $\sqrt{N}$ contiguous subarrays of size $\sqrt{N}$. Sort subarrays recursively

Counts the # of random accesses
Distribution sort outline

2. Choose \( \sqrt{N} \) "good" pivots \( p_1 \leq p_2 \leq \ldots \leq p_{\sqrt{N} - 1} \).

3. Distribute subarrays into \textit{buckets}, according to pivots

\[
\text{Bucket 1} \quad \text{Bucket 2} \quad \ldots \quad \text{Bucket } \sqrt{N}
\]

\( \sqrt{N} \leq p_1 \leq p_2 \leq \ldots \leq p_{\sqrt{N} - 1} \leq \sqrt{N} \leq \text{size} \leq 2 \sqrt{N} \)

Distribution sort analysis sketch

- Step 1 (implicitly) divides array and sorts \( \sqrt{N} \) size-\( \sqrt{N} \) subproblems
- Step 4 sorts \( \sqrt{N} \) buckets of size \( \sqrt{N} \leq n_i \leq 2 \sqrt{N} \), with total size \( N \)
- Step 5 copies back the output, with a scan

\[ T(N) = \sqrt{N} T(\sqrt{N}) + \sum T(n_i) + \Theta(N/B) + \text{Step 2&3} \]
\[ \approx 2\sqrt{N} T(\sqrt{N}) + \Theta(N/B) \]
Base: \( T(M) = 1 \)
\[ = \Theta((N/B) \log_{M/B} (N/B)) \text{ if } M \geq B \]

Missing steps

2. Choose \( \sqrt{N} \) "good" pivots \( p_1 \leq p_2 \leq \ldots \leq p_{\sqrt{N} - 1} \).
   (2) Not too hard in \( \Theta(N/B) \)

3. Distribute subarrays into \textit{buckets}, according to pivots

\[
\text{Bucket 1} \quad \text{Bucket 2} \quad \ldots \quad \text{Bucket } \sqrt{N}
\]

\( \sqrt{N} \leq p_1 \leq p_2 \leq \ldots \leq p_{\sqrt{N} - 1} \leq \sqrt{N} \leq \text{size} \leq 2 \sqrt{N} \)
Naïve distribution

- Distribute first subarray, then second, then third, ...
- Cost is only $O(N/B)$ to scan input array
- What about writing to the output buckets?
  - Suppose each subarray writes 1 element to each bucket. Cost is 1 I/O per write, for $N$ total!

Better recursive distribution

Given subarrays $s_1, ..., s_k$ and buckets $b_1, ..., b_k$
1. Recursively distribute $s_1, ..., s_{k/2}$ to $b_1, ..., b_{k/2}$
2. Recursively distribute $s_{k/2}, ..., s_k$ to $b_{k/2}, ..., b_k$

Despite crazy order, each subarray operates left to right. So only need to check next pivot.

Distribute analysis

Counting only "random accesses" here
- $D(k) = 4D(k/2) + O(k)$

Base case: when the next block in each of the $k$ buckets/subarrays fits in memory
  (this is like an $M/B$-way merge)
- So we have $D(M/B) = D(B) = \text{free}$

Solves to $D(k) = O(k^2/B)$
⇒ distribute uses $O(N/B)$ random accesses — the rest is scanning at a cost of $O(1/B)$ per element

Note on distribute

If you unroll the recursion, it's going in Z-morton order on this matrix:

- i.e., first distribute $s_1$ to $b_1$, then $s_2$ to $b_2$, then $s_3$ to $b_1$, then $s_2$ to $b_2$, and so on.