Part 2, course 2: Cache Oblivious Algorithms

CR10: Data Aware Algorithms

October 2, 2019

Agenda

Previous course (Sep. 25):

- Ideal Cache Model and External Memory Algorithms

Today:

- Cache Oblivious Algorithms and Data Structures

Next week (Oct. 9):

- Parallel External Memory Algorithms
- Parallel Cache Oblivious Algorithms:
  Multithreaded Computations

The week after (Oct. 16):

- Test (∼1.5h)

(on pebble games, external memory and cache oblivious algorithms)

- Presentation of the projects

NB: no course on Oct. 25.

Outline

Outline

Motivation for Cache-Oblivious Algorithms

I/O-optimal algorithms in the external memory model:
Depend on the memory parameters B and M: cache-aware

- Blocked-Matrix-Product: block size $b = \sqrt{M}/3$
- Merge-Sort: $K = M/B - 1$
- B-Trees: degree of a node in $O(B)$

Goal: design I/O-optimal algorithms that do not known M and B

- Self-tuning
- Optimal for any cache parameters
  $\rightarrow$ optimal for any level of the cache hierarchy!

Ideal cache model:

- Ideal-cache model
- No explicit operations on blocks as in EM

Outline

Main Tool: Divide and Conquer

Major tool:

- Split problem into smaller sizes
- At some point, size gets smaller than the cache size: no I/O needed for next recursive calls
- Analyse I/O for these "leaves" of the recursion tree and divide/merge operations

Example: Recursive matrix multiplication:

$$A = \left(\begin{array}{c|c} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{array}\right) B = \left(\begin{array}{c|c} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{array}\right) C = \left(\begin{array}{c|c} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{array}\right)$$

- If $N > 1$, compute:
  $C_{1,1} = RecMatMult(A_{1,1}, B_{1,1}) + RecMatMult(A_{1,2}, B_{2,1})$
  $C_{1,2} = RecMatMult(A_{1,1}, B_{1,2}) + RecMatMult(A_{1,2}, B_{2,2})$
  $C_{2,1} = RecMatMult(A_{2,1}, B_{1,1}) + RecMatMult(A_{2,2}, B_{2,1})$
  $C_{2,2} = RecMatMult(A_{2,1}, B_{1,2}) + RecMatMult(A_{2,2}, B_{2,2})$
- Base case: multiply elements

Main Tool: Divide and Conquer

Major tool:

- Split problem into smaller sizes
- At some point, size gets smaller than the cache size: no I/O needed for next recursive calls
- Analyse I/O for these "leaves" of the recursion tree and divide/merge operations

Example: Recursive matrix multiplication:

$$A = \left(\begin{array}{c|c} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{array}\right) B = \left(\begin{array}{c|c} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{array}\right) C = \left(\begin{array}{c|c} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{array}\right)$$

- If $N > 1$, compute:
  $C_{1,1} = RecMatMult(A_{1,1}, B_{1,1}) + RecMatMult(A_{1,2}, B_{2,1})$
  $C_{1,2} = RecMatMult(A_{1,1}, B_{1,2}) + RecMatMult(A_{1,2}, B_{2,2})$
  $C_{2,1} = RecMatMult(A_{2,1}, B_{1,1}) + RecMatMult(A_{2,2}, B_{2,1})$
  $C_{2,2} = RecMatMult(A_{2,1}, B_{1,2}) + RecMatMult(A_{2,2}, B_{2,2})$
- Base case: multiply elements

Recursive Matrix Multiply: Analysis

$$C_{1,1} = RecMatMult(A_{1,1}, B_{1,1}) + RecMatMult(A_{1,2}, B_{2,1})$$
$$C_{1,2} = RecMatMult(A_{1,1}, B_{1,2}) + RecMatMult(A_{1,2}, B_{2,2})$$
$$C_{2,1} = RecMatMult(A_{2,1}, B_{1,1}) + RecMatMult(A_{2,2}, B_{2,1})$$
$$C_{2,2} = RecMatMult(A_{2,1}, B_{1,2}) + RecMatMult(A_{2,2}, B_{2,2})$$

Analysis:

- 8 recursive calls on matrices of size $N/2 \times N/2$
- Number of I/O for size $N \times N$: $T(N) = 8T(N/2)$
- Base case: when 3 blocks fit in the cache: $3N^2 \leq M$ no more I/O for smaller sizes, then $T(N) = O(N^2/B) = O(M/B)$
- No cost on merge, all I/O cost on leaves
- Height of the recursive call tree: $h = \log_2(N/(\sqrt{M}/3))$
- Total I/O cost:

$$T(N) = O(8^h M/B) = O(N^3/(B\sqrt{M}))$$

- Same performance as blocked algorithm!

Recursive Matrix Multiply: Analysis

$$\begin{array}{ll} RecMatMultAdd(A_{1,1}, B_{1,1}, C_{1,1}); & RecMatMultAdd(A_{1,2}, B_{2,1}, C_{1,1})) \\ RecMatMultAdd(A_{1,1}, B_{1,2}, C_{1,2}); & RecMatMultAdd(A_{1,2}, B_{2,2}, C_{1,2})) \\ RecMatMultAdd(A_{2,1}, B_{1,1}, C_{2,1}); & RecMatMultAdd(A_{2,2}, B_{2,1}, C_{2,1})) \\ RecMatMultAdd(A_{2,1}, B_{1,2}, C_{2,2}); & RecMatMultAdd(A_{2,2}, B_{2,2}, C_{2,2})) \end{array}$$

Analysis:

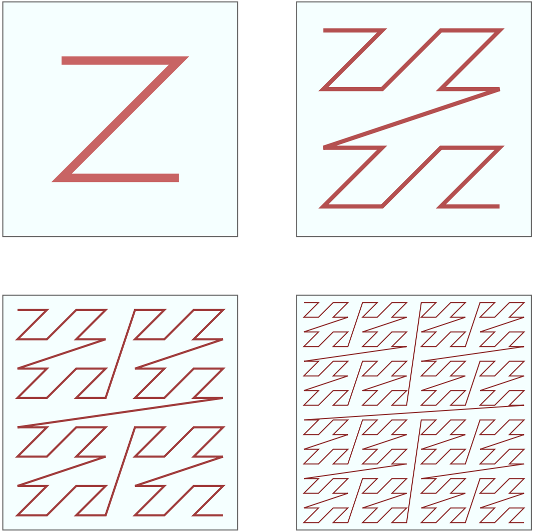
- 8 recursive calls on matrices of size $N/2 \times N/2$
- Number of I/O for size $N \times N$: $T(N) = 8T(N/2)$
- Base case: when 3 blocks fit in the cache: $3N^2 \leq M$ no more I/O for smaller sizes, then $T(N) = O(N^2/B) = O(M/B)$
- No cost on merge, all I/O cost on leaves
- Height of the recursive call tree: $h = \log_2(N/(\sqrt{M}/3))$
- Total I/O cost:

$$T(N) = O(8^h M/B) = O(N^3/(B\sqrt{M}))$$

- Same performance as blocked algorithm!

Recursive Matrix Layout

NB: previous analysis need tall-cache assumption ($M \geq B^2$)
If not, use recursive layout, e.g. bit-interleaved layout:



Recursive Matrix Layout

NB: previous analysis need tall-cache assumption ($M \geq B^2$)

If not, use recursive layout, e.g. bit-interleaved layout:

y: \ x:	0 000	1 001	2 010	3 011	4 100	5 101	6 110	7 111
0 000	000000	000001	000100	000101	010000	010001	010100	010101
1 001	000010	000011	000110	000111	010010	010011	010110	010111
2 010	001000	001001	001100	001101	011000	011001	011100	011101
3 011	001010	001011	001110	001111	011010	011011	011110	011111
4 100	100000	100001	100100	100101	110000	110001	110100	110101
5 101	100010	100011	100110	100111	110010	110011	110110	110111
6 110	101000	101001	101100	101101	111000	111001	111100	111101
7 111	101010	101011	101110	101111	111010	111011	111110	111111

Recursive Matrix Layout

NB: previous analysis need tall-cache assumption ($M \geq B^2$)
If not, use recursive layout, e.g. bit-interleaved layout:

Also known as the Z-Morton layout

Other recursive layouts:

- U-Morton, X-Morton, G-Morton
- Hilbert layout

Address computations may become expensive ☹

Possible mix of classic tiles/recursive layout

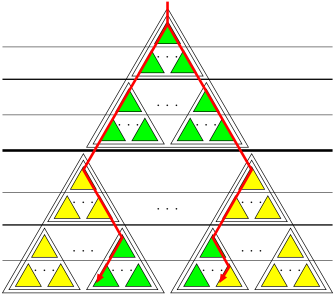
Outline

Static Search Trees

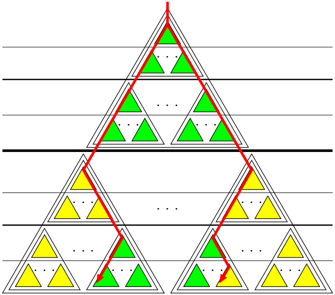
Problem with B-trees: degree depends on B ☹

Binary search tree with recursive layout:

- Complete binary search tree with N nodes (one node per element)
- Stored in memory using recursive "van Emde Boas" layout:
  - Split the tree at the middle height
  - Top subtree of size $\sim \sqrt{N} \to$ recursive layout
  - $\sim \sqrt{N}$ subtrees of size $\sim \sqrt{N} \to$ recursive layout
  - If height h is not a power of 2, set subtree height to $2^{\lceil \log_2 h \rceil} = [\![h]\!]$



Static Search Trees – Analysis



I/O complexity of search operation:

- For simplicity, assume N is a power of two
- For some height h, a subtree fits in one block ($B \approx 2^h$)
- Reading such a subtree requires at most 2 blocks
- Root-to-leaf path of length $\log_2 N$
- I/O complexity: $O(\log_2 N / \log_2 B) = O(\log_B N)$
- Meets the lower bound ☺
- Only static data-structure ☹

Outline

Cache-Oblivious Sorting: Funnels

- Binary Merge Sort: cache-oblivious ☺, not I/O optimal ☹
- K-way MergeSort: depends on M and B ☹, I/O optimal ☺

New data-structure: K-funnel

- Complete binary tree with K leaves
- Stored using van Emde Boas layout
- Buffer of size $K^{3/2}$ between each subtree and the topmost part (total: K^2 in these buffers)
- Each recursive subtree is a $\sqrt{K}$-funnel

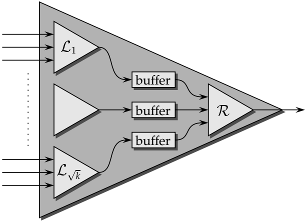
Total storage in a K funnel: $\Theta(K^2)$
(storage $S(K) = K^2 + (1+\sqrt{K})S(\sqrt{K})$)

Cache-Oblivious Sorting: Funnels

- Binary Merge Sort: cache-oblivious ☺, not I/O optimal ☹
- K-way MergeSort: depends on M and B ☹, I/O optimal ☺

New data-structure: K-funnel

- Complete binary tree with K leaves
- Stored using van Emde Boas layout
- Buffer of size $K^{3/2}$ between each subtree and the topmost part (total: K^2 in these buffers)
- Each recursive subtree is a $\sqrt{K}$-funnel



Total storage in a K funnel: $\Theta(K^2)$
(storage $S(K) = K^2 + (1 + \sqrt{K})S(\sqrt{K})$)

Lazy Funnels

- Consider resulting tree where edges are buffers
- Output buffer of a K-funnel has size K^3

Fill algorithm: while output buffer not full

1. If left input buffer empty, call Fill on left child
2. If right input buffer empty, call Fill on right child
3. Perform one merge step:
   Move smallest of left and right buffers (front) to output (rear)

- Buffer exhaustion propagates upward
- I/O complexity of filling output buffer of size K^3:

$$O\left(\frac{K^3}{B}\log_M K^3 + K\right)$$

Funnel Sort

Funnel-Sort

1. Split input in $N^{1/3}$ segments of size $N^{2/3}$
2. Sort segments recursively
3. Use funnel with $K = N^{1/3}$ to produce output

I/O complexity: $O(SORT(N))$
Nb of comparisons: $O(N \log N)$

Analysis (big picture):

- Some J-funnel fits in cache, together with its input buffers
- When input buffer empty, J-funnel may be flushed to memory
- Bound the number of flushes and I/Os in the funnel

Outline

PMA: Packed Memory Array

Goal: Store N ordered elements in array of size $P = cN$

Contradictory objectives:

- Pack nodes for fast scan of S elements ($O(1 + S/B)$)
- Leave enough room for fast insertion

PMA:

- Array divided into segments of size $\log P$
- Virtual complete binary tree whose leaves are these segments
- Density of a node:

$$\rho = \frac{\text{number of elements in subtree}}{\text{capacity of the subtree}}$$

- Constraints on density: $1/2 - 1/4d/h \leq \rho \leq 1/2 + 1/4d/h$
  d: depth of the node, h: height of the tree
  $\rightarrow$ up in the tree: less slack on density

Packed Memory Array: Details

Insertion algorithm:

- Find segment (leaf in the tree)
- If segment not full, insert (move other elements if needed)
- If segment full, before inserting the element:
  1. Climb in tree to find ancestor that respects density constraints
     Parallel left and right scan counting elements
  2. Rebalance subtree: redistribute all elements uniformly in existing leafs
     Some additional scans
  3. For big changes in N: rebuild everything

(Same for deletions)
Amortized cost of insertion: $O(1 + \log^2 N/B)$

Packed Memory Array: Details

Insertion algorithm:

- Find segment (leaf in the tree)
- If segment not full, insert (move other elements if needed)
- If segment full, before inserting the element:
  1. Climb in tree to find ancestor that respects density constraints
     Parallel left and right scan counting elements
  2. Rebalance subtree: redistribute all elements uniformly in existing leafs
     Some additional scans
  3. For big changes in N: rebuild everything

(Same for deletions)
Amortized cost of insertion: $O(1 + \log^2 N/B)$

Cache-Oblivious B-Trees

- PMA to store elements
- Static Search Tree with $\Theta(N)$ leaves
  a node store the maximum of its two children
- Bi-directional pointers between tree leaves and PMA cells

- Search: using search tree
- Insertion: in the PMA, then propagate changes in the tree

Theorem (Cache-oblivious B-Trees).

This data-structure has the following I/O cost:

- Insertion and deletions in $O(\log_B N + (\log^2 N)/B)$ (amortized)
- Search in $O(\log_B N)$
- Scanning K consecutive elements in $O(\lceil K/B \rceil)$

NB: Removing the $(\log^2 N)/B$ term leads to loosing fast scanning

Outline

Distribution sweeping for geometric problem

Distribution sweeping:

- Sort geometric objects (e.g. w.r.t. one dimension)
- Split problem into strips
- Divide-and-conquer approach on strips
- Merge results via a sweep of strips in another dimension (cache-oblivious merge: 2-way)

Multidimensional Maxima Problem

Given a set of points in d dimensions, a point $p = (p_1, p_2, \ldots, p_d)$ dominates another point q if $p_i \geq q_i$ for all i. Given N points, report the maximal points (points non dominated).

- 1D: Single maximum
- 2D: Simple sweep algorithm:
  1. Sort point by decreasing first coordinate
  2. Report point if its second coordinate is larger than the one of the last reported point

Distribution sweeping for geometric problem

Distribution sweeping:

- Sort geometric objects (e.g. w.r.t. one dimension)
- Split problem into strips
- Divide-and-conquer approach on strips
- Merge results via a sweep of strips in another dimension (cache-oblivious merge: 2-way)

Multidimensional Maxima Problem

Given a set of points in d dimensions, a point $p = (p_1, p_2, \ldots, p_d)$ dominates another point q if $p_i \geq q_i$ for all i. Given N points, report the maximal points (points non dominated).

- 1D: Single maximum
- 2D: Simple sweep algorithm:
  1. Sort point by decreasing first coordinate
  2. Report point if its second coordinate is larger than the one of the last reported point

Divide-and-Conquer for 3D Maxima

- Sort points according to z
- Divide points in strips
- For each strip: report (output) maximal points sorted by decreasing x

Base case: strip with a single point (reported)
When merging strips A and B (with $z_B > z_A$):

- all points in B have larger z: all maximal points kept
- maximal points in A are maximal in $A \cup B$ iff there are not dominated by some maximal point of B

Merging Algorithm:

- Scan maximal points of A and B by decreasing x
- Keep track of the largest y_B of nodes from B
- If next node comes from B: keep it (output), update y_B
- If next node comes from A and has larger y than current y_B: keep it (output), otherwise, delete it

Divide-and-Conquer for 3D Maxima

- Sort points according to z
- Divide points in strips
- For each strip: report (output) maximal points sorted by decreasing x

Base case: strip with a single point (reported)
When merging strips A and B (with $z_B > z_A$):

- all points in B have larger z: all maximal points kept
- maximal points in A are maximal in $A \cup B$ iff there are not dominated by some maximal point of B

Merging Algorithm:

- Scan maximal points of A and B by decreasing x
- Keep track of the largest y_B of nodes from B
- If next node comes from B: keep it (output), update y_B
- If next node comes from A and has larger y than current y_B: keep it (output), otherwise, delete it

Divide-and-Conquer for 3D Maxima

- Sort points according to z
- Divide points in strips
- For each strip: report (output) maximal points sorted by decreasing x

Base case: strip with a single point (reported)
When merging strips A and B (with $z_B > z_A$):

- all points in B have larger z: all maximal points kept
- maximal points in A are maximal in $A \cup B$ iff there are not dominated by some maximal point of B

Merging Algorithm:

- Scan maximal points of A and B by decreasing x
- Keep track of the largest y_B of nodes from B
- If next node comes from B: keep it (output), update y_B
- If next node comes from A and has larger y than current y_B: keep it (output), otherwise, delete it

Divide-and-Conquer for 3D Maxima

Summary of algorithm:

1. Sort by decreasing z
2. Recursively process strips
3. Merge sorted sequences by comparing to y_B, remove some nodes

Cache-oblivious version:

- Step 1: (Lazy) Funnel-Sort
- Step 3: (Lazy) Funnel-Sort with modified merger to remember y_B
  ($O(1)$ extra space on each node)

Complexity: $O(SORT(N))$

Divide-and-Conquer for 3D Maxima

Summary of algorithm:

1. Sort by decreasing z
2. Recursively process strips
3. Merge sorted sequences by comparing to y_B, remove some nodes

Cache-oblivious version:

- Step 1: (Lazy) Funnel-Sort
- Step 3: (Lazy) Funnel-Sort with modified merger to remember y_B
  ($O(1)$ extra space on each node)

Complexity: $O(SORT(N))$

Outline

Conclusion

- Clean model, algorithms independent from architectural parameters M and B
- Good news: match external memory bounds in most cases

- Best tool: divide-and-conquer
- Base case of the analysis differs from algorithm base case:
  - Sometimes $N = \Theta(M)$ (mergesort, matrix mult.,...)
  - Sometimes $N\Theta(B)$ (static search tree, ...)

- New algorithmic solutions to force data locality
- Can lead to real performance gains for large N