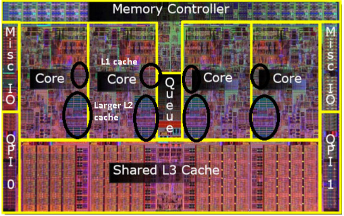
Data Aware Algorithms – Part 2
Cache-Oblivious Algorithms

Loris Marchal

October 18, 2023

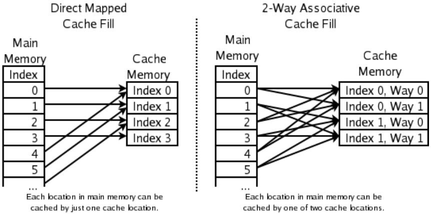
Outline

Properties of real caches



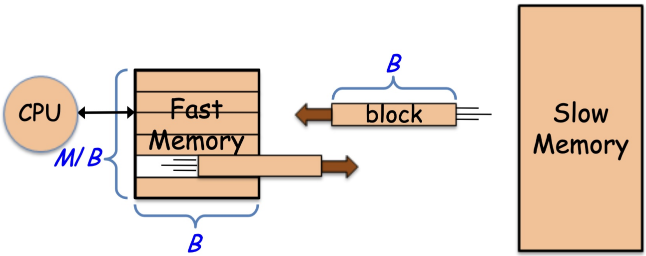
- Memory/cache divided into blocks (or lines or pages) of size B
- When requested data not in cache (cache miss), corresponding block automatically loaded
- Limited associativity:
  - each block of memory belongs to a cluster (usually computed as *address* % M)
  - at most c blocks of a cluster can be stored in cache at once (c-way associative)
  - Trade-off between hit rate and time for searching the cache
- If cache full, blocks have to be evicted: Standard block replacement policy: Least Recently Used (LRU)

Properties of real caches



- Memory/cache divided into blocks (or lines or pages) of size B
- When requested data not in cache (cache miss), corresponding block automatically loaded
- Limited associativity:
  - each block of memory belongs to a cluster (usually computed as *address* % M)
  - at most c blocks of a cluster can be stored in cache at once (c-way associative)
  - Trade-off between hit rate and time for searching the cache
- If cache full, blocks have to be evicted: Standard block replacement policy: Least Recently Used (LRU)

Ideal cache model



- Fully associative
  $c = \infty$, blocks can be store everywhere in the cache
- Optimal replacement policy
  Belady's rule: evict block whose next access is furthest
- Tall cache: $M/B \gg B \qquad (M = \Theta(B^2))$

LRU vs. Optimal Replacement Policy

replacement policy	cache size	nb of cache misses
LRU	k_{LRU}	$T_{LRU}(s)$
OPT	$k_{\text{OPT}} \leq k_{\text{LRU}}$	$T_{OPT}(s)$

OPT: optimal (offline) replacement policy (Belady's rule)

Theorem (Sleator and Tarjan, 1985).

For any sequence s:

$$T_{\text{LRU}}(s) \leq \frac{k_{\text{LRU}}}{k_{\text{LRU}} - k_{\text{OPT}} + 1}(T_{\text{OPT}}(s) + k_{\text{OPT}})$$

If LRU cache initially contains all pages in OPT cache: remove the additive term

Theorem (Bound on competitive ratio).

Assume there exists a and b such that $T_A(s) \leq aT_{\text{OPT}}(s) + b$ for all s, then $a \geq k_A/(k_A - k_{\text{OPT}} + 1)$.

LRU vs. Optimal Replacement Policy

replacement policy	cache size	nb of cache misses
LRU	k_{LRU}	$T_{LRU}(s)$
OPT	$k_{\text{OPT}} \leq k_{\text{LRU}}$	$T_{OPT}(s)$

OPT: optimal (offline) replacement policy (Belady's rule)

Theorem (Sleator and Tarjan, 1985).

For any sequence s:

$$T_{\text{LRU}}(s) \leq \frac{k_{\text{LRU}}}{k_{\text{LRU}} - k_{\text{OPT}} + 1}(T_{\text{OPT}}(s) + k_{\text{OPT}})$$

If LRU cache initially contains all pages in OPT cache: remove the additive term

Theorem (Bound on competitive ratio).

Assume there exists a and b such that $T_A(s) \leq aT_{\text{OPT}}(s) + b$ for all s, then $a \geq k_A/(k_A - k_{\text{OPT}} + 1)$.

LRU vs. Optimal Replacement Policy

replacement policy	cache size	nb of cache misses
LRU	k_{LRU}	$T_{LRU}(s)$
OPT	$k_{\mathrm{OPT}} \leq k_{\mathrm{LRU}}$	$T_{OPT}(s)$

OPT: optimal (offline) replacement policy (Belady's rule)

Theorem (Sleator and Tarjan, 1985).

For any sequence s:

$$T_{\mathrm{LRU}}(s) \leq \frac{k_{\mathrm{LRU}}}{k_{\mathrm{LRU}} - k_{\mathrm{OPT}} + 1}(T_{\mathrm{OPT}}(s) + k_{\mathrm{OPT}})$$

If LRU cache initially contains all pages in OPT cache:
remove the additive term

Theorem (Bound on competitive ratio).

Assume there exists a and b such that $T_A(s) \leq a T_{\mathrm{OPT}}(s) + b$ for all s, then $a \geq k_A/(k_A - k_{\mathrm{OPT}} + 1)$.

LRU competitive ratio – Proof

- Consider any subsequence t of s, such that $T_{\text{LRU}}(t) \leq k_{\text{LRU}}$ (t should not include first request)
- Let p_i be the block request right before t in s
- If LRU loads twice the same block in s, then $T_{\text{LRU}}(t) \geq k_{\text{LRU}} + 1$ (contradiction)
- Same if LRU loads p_i during t
- Thus on t, LRU loads $T_{\text{LRU}}(t)$ different blocks, different from p_i
- When starting t, OPT has p_i in cache
- On t, OPT must load at least $T_{\text{LRU}}(t) - k_{\text{OPT}} + 1$

- Partition s into $s_0, s_1, \ldots, s_n$ such that $T_{\text{LRU}}(s_0) \leq k_{\text{LRU}}$ and $T_{\text{LRU}}(s_i) = k_{\text{LRU}}$ for $i > 1$
- On s_0, $T_{\text{OPT}}(s_0) \geq T_{\text{LRU}}(s_0) - k_{\text{OPT}}$
- In total for LRU: $T_{\text{LRU}} = T_{\text{LRU}}(s_0) + nk_{\text{LRU}}$
- In total for OPT: $T_{\text{OPT}} \geq T_{\text{LRU}}(s_0) - k_{\text{OPT}} + n(k_{\text{LRU}} - k_{\text{OPT}} + 1)$

Bound on Competitive Ratio – Proof

Consider any online algorithm A:

- Let S_A^{init} (resp. $S_{\text{OPT}}^{\text{init}}$) the set of blocks initially in A'cache (resp. OPT's cache)
- Consider the block request sequence made of two steps:

  S_1: $k_A - k_{\text{OPT}} + 1$ (new) blocks not in $S_A^{\text{init}} \cup S_{\text{OPT}}^{\text{init}}$
  S_2: $k_{\text{OPT}} - 1$ blocks s.t. then next block is always in $(S_{\text{OPT}}^{\text{init}} \cup S_1) \backslash S_A$

  NB: step 2 is possible since $|S_{\text{OPT}}^{\text{init}} \cup S_1| = k_A + 1$

- A loads one block for each request of both steps: k_A loads
- OPT loads one block only in S_1: $k_A - k_{\text{OPT}} + 1$ loads

NB: Repeat this process to create arbitrarily long sequences.

Justification of the Ideal Cache Model

Theorem (Frigo et al, 1999).

If an algorithm makes T memory transfers with a cache of size $M/2$ with optimal replacement, then it makes at most $2T$ transfers with cache size M with LRU.

Definition (Regularity condition).

Let $T(M)$ be the number of memory transfers for an algorithm with cache of size M and an optimal replacement policy. The regularity condition of the algorithm writes

$$T(M) = O(T(M/2))$$

Corollary

If an algorithm follows the regularity condition and makes $T(M)$ transfers with cache size M and an optimal replacement policy, it makes $\Theta(T(M))$ memory transfers with LRU.

Justification of the Ideal Cache Model

Theorem (Frigo et al, 1999).

If an algorithm makes T memory transfers with a cache of size $M/2$ with optimal replacement, then it makes at most $2T$ transfers with cache size M with LRU.

Definition (Regularity condition).

Let $T(M)$ be the number of memory transfers for an algorithm with cache of size M and an optimal replacement policy. The regularity condition of the algorithm writes

$$T(M) = O(T(M/2))$$

Corollary

If an algorithm follows the regularity condition and makes $T(M)$ transfers with cache size M and an optimal replacement policy, it makes $\Theta(T(M))$ memory transfers with LRU.

Outline

Ideal Cache Model

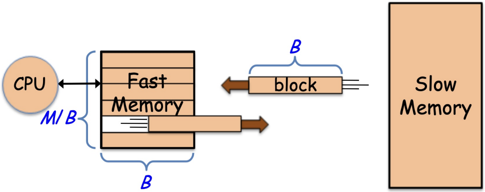
External Memory Algorithms and Data Structures

- External Memory Model
- Merge Sort
- Searching and B-Trees

Cache Oblivious Algorithms and Data Structures

- Motivation
- Divide and Conquer
- Static Search Trees

External Memory Model



- External Memory: storage (large)
- Internal Memory: for computations, size M
- Ideal cache model for transfers: blocks of size B
- Input size of the problem: N
- Main metric: number of blocks moved from/to the cache

Basic operation

Scanning N elements stored in a contiguous segment of memory costs at most $\lceil N/B \rceil + 1$ memory transfers.

Merge Sort in External Memory

Standard Merge Sort: Divide and Conquer

1. Recursively split the array (size N) in two, until reaching size 1
2. Merge two sorted arrays of size L into one of size $2L$ requires $2L$ comparisons

In total: $\log N$ levels, N comparisons in each level

Adaptation for External Memory: Phase 1

- Partition the array in N/M chunks of size M
- Sort each chunks independently ($\rightarrow$ runs)
- Block transfers: $2M/B$ per chunk, $2N/B$ in total
- Number of comparisons: $M \log M$ per chunk, $N \log M$ in total

Merge Sort in External Memory

Standard Merge Sort: Divide and Conquer

1. Recursively split the array (size N) in two, until reaching size 1
2. Merge two sorted arrays of size L into one of size $2L$ requires $2L$ comparisons

In total: $\log N$ levels, N comparisons in each level

Adaptation for External Memory: Phase 1

- Partition the array in N/M chunks of size M
- Sort each chunks independently ($\rightarrow$ runs)
- Block transfers: $2M/B$ per chunk, $2N/B$ in total
- Number of comparisons: $M \log M$ per chunk, $N \log M$ in total

Two-Way Merge in External Memory

Phase 2:

Merge two runs R and S of size $L \rightarrow$ one run T of size $2L$

1. Load first blocks $\widehat{R}$ (and $\widehat{S}$) of R (and S)
2. Allocate first block $\widehat{T}$ of T
3. While R and S both not exhausted
   (a) Merge as much $\widehat{R}$ and $\widehat{S}$ into $\widehat{T}$ as possible
   (b) If $\widehat{R}$ (or $\widehat{S}$) gets empty, load new block of R (or S)
   (c) If $\widehat{T}$ gets full, flush it into T
4. Transfer remaining items of R (or S) in T

- Internal memory usage: 3 blocks
- Block transfers: $2L/B$ reads $+$ $2L/B$ writes $= 4L/B$
- Number of comparisons: $2L$

Two-Way Merge in External Memory

Phase 2:

Merge two runs R and S of size $L \to$ one run T of size $2L$

1. Load first blocks $\widehat{R}$ (and $\widehat{S}$) of R (and S)
2. Allocate first block $\widehat{T}$ of T
3. While R and S both not exhausted
   (a) Merge as much $\widehat{R}$ and $\widehat{S}$ into $\widehat{T}$ as possible
   (b) If $\widehat{R}$ (or $\widehat{S}$) gets empty, load new block of R (or S)
   (c) If $\widehat{T}$ gets full, flush it into T
4. Transfer remaining items of R (or S) in T

- Internal memory usage: 3 blocks
- Block transfers: $2L/B$ reads $+$ $2L/B$ writes $= 4L/B$
- Number of comparisons: $2L$

Total complexity of Two-Way Merge Sort

Analysis at each level:

- At level k: runs of size $2^k M$ (nb: $N/(2^k M)$)
- Merge to reach levels $k = 1 \dots \log_2 N/M$
- Block transfers at level k: $2^{k+1} M/B \times N/(2^k M) = 2N/B$
- Number of comparisons: N

Total complexity of phases 1+2:

- Block transfers: $2N/B(1 + \log_2 N/M) = O(N/B \log_2 N/M)$
- Number of comparisons: $N \log M + N \log_2 N/M = N \log N$

- Internal memory used ?

Total complexity of Two-Way Merge Sort

Analysis at each level:

- At level k: runs of size $2^k M$ (nb: $N/(2^k M)$)
- Merge to reach levels $k = 1 \ldots \log_2 N/M$
- Block transfers at level k: $2^{k+1} M/B \times N/(2^k M) = 2N/B$
- Number of comparisons: N

Total complexity of phases 1+2:

- Block transfers: $2N/B(1 + \log_2 N/M) = O(N/B \log_2 N/M)$
- Number of comparisons: $N \log M + N \log_2 N/M = N \log N$

- Internal memory used ?

Total complexity of Two-Way Merge Sort

Analysis at each level:

- At level k: runs of size $2^k M$ (nb: $N/(2^k M)$)
- Merge to reach levels $k = 1 \dots \log_2 N/M$
- Block transfers at level k: $2^{k+1} M/B \times N/(2^k M) = 2N/B$
- Number of comparisons: N

Total complexity of phases 1+2:

- Block transfers: $2N/B(1 + \log_2 N/M) = O(N/B \log_2 N/M)$
- Number of comparisons: $N \log M + N \log_2 N/M = N \log N$

- Internal memory used ?

Total complexity of Two-Way Merge Sort

Analysis at each level:

- At level k: runs of size $2^k M$ (nb: $N/(2^k M)$)
- Merge to reach levels $k = 1 \dots \log_2 N/M$
- Block transfers at level k: $2^{k+1}M/B \times N/(2^k M) = 2N/B$
- Number of comparisons: N

Total complexity of phases 1+2:

- Block transfers: $2N/B(1 + \log_2 N/M) = O(N/B \log_2 N/M)$
- Number of comparisons: $N \log M + N \log_2 N/M = N \log N$

- Internal memory used ? only 3 blocks ☹

Optimization: K-Way Merge Sort

- Consider K input runs at each merge step
- Efficient merging, e.g.: MinHeap data structure
  insert, extract: $O(\log K)$
- Complexity of merging K runs of length L: $KL \log K$
- Block transfers: no change ($2KL/B$)

Total complexity of merging:

- Block transfers: $\log_K N/M$ steps $\rightarrow 2N/B \log_K N/M$
- Computations: $N \log K$ per step $\rightarrow N \log K \times \log_K N/M$
  $= N \log_2 N/M$ (id.)

Maximize K to reduce transfers:

- $(K+1)B = M$ (K input blocks + 1 output block)
- Block transfers: $O\left(\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{M}\right)$
- NB: $\log_{M/B} N/M = \log_{M/B} N/B - 1$
- Block transfers: $O\left(\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B}\right)$

Optimization: K-Way Merge Sort

- Consider K input runs at each merge step
- Efficient merging, e.g.: MinHeap data structure
  insert, extract: $O(\log K)$
- Complexity of merging K runs of length L: $KL \log K$
- Block transfers: no change ($2KL/B$)

Total complexity of merging:

- Block transfers: $\log_K N/M$ steps $\to 2N/B \log_K N/M$
- Computations: $N \log K$ per step $\to N \log K \times \log_K N/M$
  $= N \log_2 N/M$ (id.)

Maximize K to reduce transfers:

- $(K+1)B = M$ (K input blocks + 1 output block)
- Block transfers: $O\left(\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{M}\right)$
- NB: $\log_{M/B} N/M = \log_{M/B} N/B - 1$
- Block transfers: $O\left(\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B}\right)$

Optimization: K-Way Merge Sort

- Consider K input runs at each merge step
- Efficient merging, e.g.: MinHeap data structure
  insert, extract: $O(\log K)$
- Complexity of merging K runs of length L: $KL \log K$
- Block transfers: no change $(2KL/B)$

Total complexity of merging:

- Block transfers: $\log_K N/M$ steps $\rightarrow 2N/B \log_K N/M$
- Computations: $N \log K$ per step $\rightarrow N \log K \times \log_K N/M$
  $= N \log_2 N/M$ (id.)

Maximize K to reduce transfers:

- $(K+1)B = M$ (K input blocks $+ 1$ output block)
- Block transfers: $O\left(\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{M}\right)$
- NB: $\log_{M/B} N/M = \log_{M/B} N/B - 1$
- Block transfers: $O\left(\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B}\right)$

B-Trees

- Problem: Search for a particular element in a huge dataset
- Solution: Search tree with large degree ($\approx B$)

Definition (B-tree with minimum degree d).

Search tree such that:

- Each node (except the root) has at least d children
- Each node has at most $2d - 1$ children
- Node with k children has $k - 1$ keys separating the children
- All leaves have the same depth

Proposed by Bayer and McCreigh (1972)

Search and Insertion in B-Trees

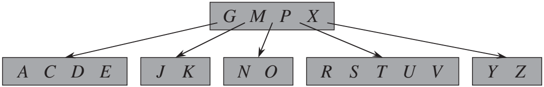
Usually, we require that $d = O(B)$

Lemma.

Searching in a B-Tree requires $O(\log_d N)$ I/Os.

Recursive algorithm for insertion of new key:

1. If root node of current subtree is full ($2d$ children), split it:
   (a) Find median key, send it to the father f (if any, otherwise it becomes the new root)
   (b) Keys and subtrees $<$ median key $\rightarrow$ new left subtree of f
   (c) Keys and subtrees $>$ median key $\rightarrow$ new right subtree f
2. If root node of current subtree $=$ leaf, insert new key
3. Otherwise, find correct subtree s, insert recursively in s



NB: height changes only when root is split $\rightarrow$ balanced tree
Number of transfers: $O(height)$

Search and Insertion in B-Trees

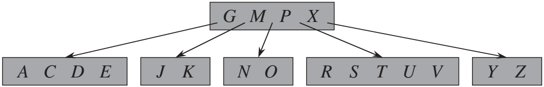
Usually, we require that $d = O(B)$

Lemma.

Searching in a B-Tree requires $O(\log_d N)$ I/Os.

Recursive algorithm for insertion of new key:

1. If root node of current subtree is full ($2d$ children), split it:
   (a) Find median key, send it to the father f
   (if any, otherwise it becomes the new root)
   (b) Keys and subtrees $<$ median key $\rightarrow$ new left subtree of f
   (c) Keys and subtrees $>$ median key $\rightarrow$ new right subtree f
2. If root node of current subtree $=$ leaf, insert new key
3. Otherwise, find correct subtree s, insert recursively in s



NB: height changes only when root is split $\rightarrow$ balanced tree

Number of transfers: $O(\mathit{height})$

Search and Insertion in B-Trees

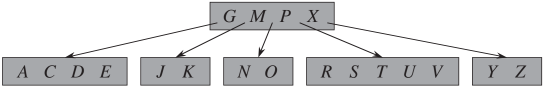
Usually, we require that $d = O(B)$

Lemma.

Searching in a B-Tree requires $O(\log_d N)$ I/Os.

Recursive algorithm for insertion of new key:

1. If root node of current subtree is full ($2d$ children), split it:
   (a) Find median key, send it to the father f
   (if any, otherwise it becomes the new root)
   (b) Keys and subtrees $<$ median key $\rightarrow$ new left subtree of f
   (c) Keys and subtrees $>$ median key $\rightarrow$ new right subtree f
2. If root node of current subtree $=$ leaf, insert new key
3. Otherwise, find correct subtree s, insert recursively in s



NB: height changes only when root is split $\rightarrow$ balanced tree
Number of transfers: $O(\mathit{height})$

Suppression in B-Trees

Suppression algorithm of k from a tree with at least d keys:

- If tree=leaf, straightforward
- If $k =$ key of internal node:
  - If subtree s immediately left of k has $\geq d$ keys, remove maximum element k' of s, replace k by k'
  - Otherwise, try the same on right subtree (with minimum)
  - Otherwise (both neighbor subtrees have $d-1$ keys): remove k and merge these neighbor subtrees
- If k is in a subtree s, suppress recursively in s
- If T has only $d-1$ keys:
  - Try to steal one key from a neighbor of T with at least d keys
  - Otherwise merge T with one of its neighbors

Number of block transfers: $O(\mathit{height})$

Usage of B-Trees

Widely used in large database and filesystems
(SQL, ext4, Apple File System, NTFS)

Variants:

- B+ Trees: store data only on leaves
  increase degree $\rightarrow$ reduce height
  add pointer from leaf to next one to speedup sequential access
- B* Trees: better balance of internal node
  (max size: $2b \rightarrow 3b/2$, nodes at least $2/3$ full)
  - When 2 siblings full: split into 3 nodes
  - Pospone splitting: shift keys to neighbors if possible

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 value of cache parameters
  $\rightarrow$ optimal for any level of the cache hierarchy!

Cache-Oblivious model:

- Ideal-cache model
- No explicit operations on blocks as in external memory algos.

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

$$
\begin{aligned}
C_{1,1} &= RecMatMult(A_{1,1}, B_{1,1}) + RecMatMult(A_{1,2}, B_{2,1}) \\
C_{1,2} &= RecMatMult(A_{1,1}, B_{1,2}) + RecMatMult(A_{1,2}, B_{2,2}) \\
C_{2,1} &= RecMatMult(A_{2,1}, B_{1,1}) + RecMatMult(A_{2,2}, B_{2,1}) \\
C_{2,2} &= RecMatMult(A_{2,1}, B_{1,2}) + RecMatMult(A_{2,2}, B_{2,2})
\end{aligned}
$$

- 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 for the analysis: 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!
- What if we choose $3N^2 = B$ as base case ?
- If I/Os not only on leaves:
use Master Theorem for divide-and-conquer recurrences

Recursive Matrix Multiply: Analysis

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

- 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 for the analysis: 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!
- What if we choose $3N^2 = B$ as base case ?
- If I/Os not only on leaves: use Master Theorem for divide-and-conquer recurrences

Recursive Matrix Multiply: Analysis

$RecMatMultAdd(A_{1,1}, B_{1,1}, C_{1,1})$; $RecMatMultAdd(A_{1,2}, B_{2,1}, C_{1,1}))$
$RecMatMultAdd(A_{1,1}, B_{1,2}, C_{1,2})$; $RecMatMultAdd(A_{1,2}, B_{2,2}, C_{1,2}))$
$RecMatMultAdd(A_{2,1}, B_{1,1}, C_{2,1})$; $RecMatMultAdd(A_{2,2}, B_{2,1}, C_{2,1}))$
$RecMatMultAdd(A_{2,1}, B_{1,2}, C_{2,2})$; $RecMatMultAdd(A_{2,2}, B_{2,2}, C_{2,2}))$

- 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 for the analysis: 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!
- What if we choose $3N^2 = B$ as base case ?
- If I/Os not only on leaves:
use Master Theorem for divide-and-conquer recurrences

Recursive Matrix Multiply: Analysis

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

- 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 for the analysis: 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!
- What if we choose $3N^2 = B$ as base case ?
- If I/Os not only on leaves: use Master Theorem for divide-and-conquer recurrences

Recursive Matrix Multiply: Analysis

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

- 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 for the analysis: 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!
- What if we choose $3N^2 = B$ as base case ?
- If I/Os not only on leaves:
use Master Theorem for divide-and-conquer recurrences

Recursive Matrix Multiply: Analysis

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

- 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 for the analysis: 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!
- What if we choose $3N^2 = B$ as base case ?
- If I/Os not only on leaves:
use Master Theorem for divide-and-conquer recurrences

Recursive Matrix Multiply: Analysis

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

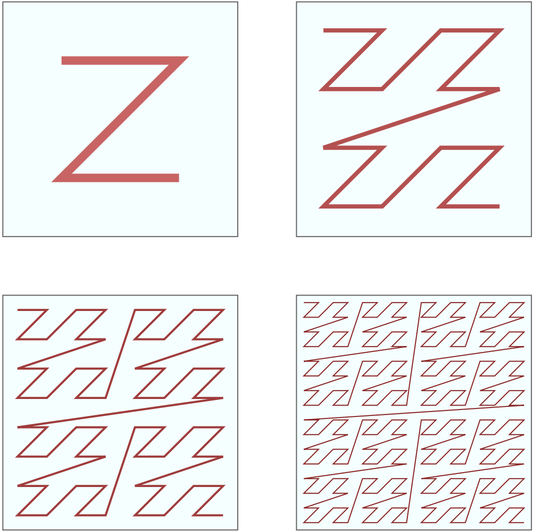
- 8 recursive calls on matrices of size $N/2 \times N/2$
- Number of I/O for size $N \times N$: $T(N) = 8T(N/2)$
- Base case for the analysis: when 3 blocks fit in the cache ($3N^2 \leq M$) no more I/O for smaller sizes, then
$$T(N) = O(N^2/B) = O(M/B)$$
- No cost on merge, all I/O cost on leaves
- Height 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!
- What if we choose $3N^2 = B$ as base case ?
- If I/Os not only on leaves:
  use Master Theorem for divide-and-conquer recurrences

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:



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 curve

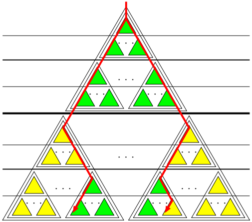
Address computations may become expensive ☹

Possible mix of classic tiles/recursive layout

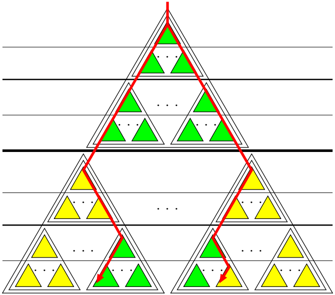
Static Search Trees

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 $2^{\lceil \log_2 h \rceil} = [\![h]\!]$
  - one subtree stored contiguously in memory (any order among subtrees)



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 ☹

Conclusion

- External memory: clean model to study blocked I/O
- To derive lower bounds and algorithms reaching these bounds
- Cache-oblivious: algorithms independent from architectural parameters M and B

- 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
- Successful implementations
  (e.g. data structures for databases)