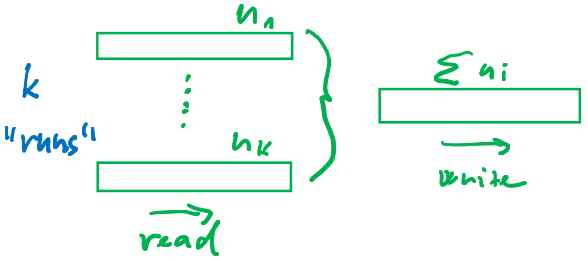
Multiway mergesort (better use of cache)



needs $\lceil \log_k N \rceil = \lceil \log N / \log k \rceil$ passes

minimum of k items by heap $\Rightarrow \Theta(\log k)$ per item

single pass $\Theta(N \log k)$ time

$\Rightarrow$ k-way mergesort in $\Theta(N \log k \cdot \log N / \log k) = \Theta(N \log N)$ time

↑ independent on k

I/O complexity:

- single merge: $\Theta(\Sigma n_i / B + 1)$ ← runs are consecutive
- single pass: $\Theta(N/B + 1)$ ← successive merges on consecutive runs
- in total: $\Theta(N/B \log N / \log k + 1)$ ← for $N < B$ when $O(1)$ suffice otherwise $O(N/B+1) = O(N/B)$

How large cache we need?

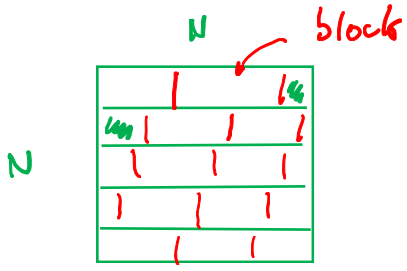
$k+1$ blocks for single merge, heap $\le k-1$ blocks $\Rightarrow$ $M \ge 2Bk$ suffice

- In I/O model + cache-aware model, set $k := \lfloor M/2B \rfloor$

  $\Rightarrow$ $\Theta(N/B \frac{\log N}{\log M/B} + 1)$ block transfers (optimal)

- In cache-oblivious model this cannot be done. (funnelsort)

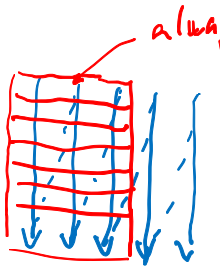
Matrix transposition (similarly matrix multiplication, ...)



row-major order

traversing the matrix

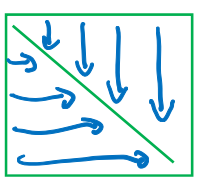
in $O(N^2/B+1)$ block transfers (scanning array)



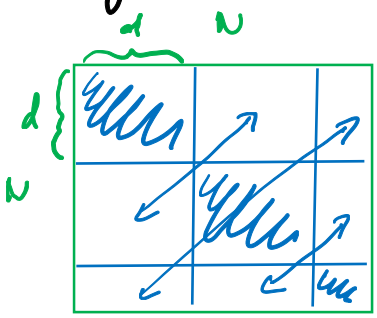
in $\Theta(N^2)$ block transfers if $M < NB$

$\Rightarrow$ naive transposition has $\Theta(N^2)$ block transfers if $M = o(NB)$

(same problem occurs in a constant fraction of columns)



Tiling (for cache-aware alg.)



tiles $d \times d$ (or smaller on borders) in $N \times N$ matrix

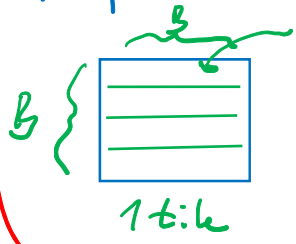
$\#\text{tiles} = \lceil N/d \rceil^2 \leq (N/d+1)^2 = O(N^2/d^2+1)$

Alg: transpose diag. tiles, transpose + swap pairs of nondiagonal tiles

Idea: if 2 tiles fit into the cache, subproblems can be solved easily.

$\Rightarrow$ set $d := B$ $\Rightarrow$ $O(N^2/B^2+1)$ tiles

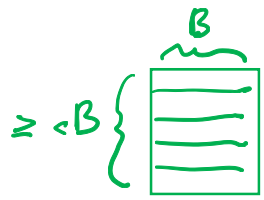
1) perfect alignment (B divides N)



each tile needs B blocks

$O(N^2/B^2)$ tiles $\Rightarrow$ $O(N^2/B)$ block transfers

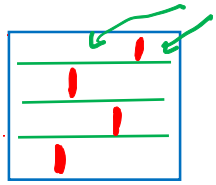
How large cache we need? $M \geq 2B^2$



"Tall cache assumption": $M = \Omega(B^2)$

if const. < 2, use tiles const. times smaller $\Rightarrow$ same asymptotics

2) not perfect alignment (B does not divide N)



each tile needs $2B$ block $\Rightarrow$ $O(B)$ transfers

$O(N^2/B^2+1)$ tiles $\Rightarrow$ $O(N^2/B+1)$ block transfers

How large cache we need? $M \geq 4B^2$

Alg. needs to know B (the tile size) $\Rightarrow$ cache-aware.

Runs in $O(N^2)$ time and $O(N^2/B+1)$ block transfers.

optimal as each nondiagonal element has to be transfered

Cache oblivious matrix transposition

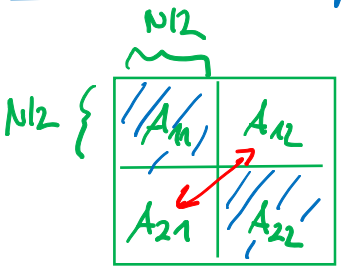
cache-aware strategy: decompose into subproblems that fit into cache

vs

cache-oblivious strategy: alg. does not know B, M but cache controller does!

- recursively subdivide to smaller and smaller problem sizes
- on some level of recursion, the cache controller employs the same replacement strategy as for cache aware alg. $\Rightarrow$ same I/O complexity (up to multiplicative constants)

Divide and conquer alg. 1) assume $N = 2^k$ for some integer k

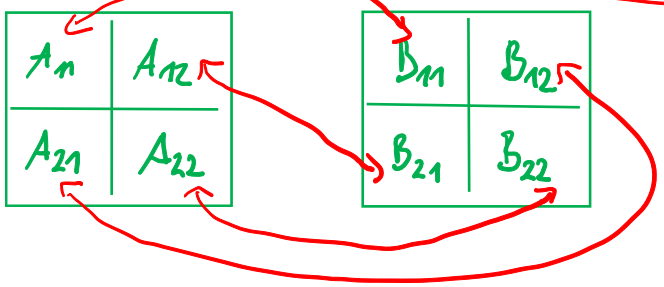


"naive" approach: transpose each submatrix + swap A_{21}, A_{12} (by (row) scanning ✓)

$T_N \to 4T_{N/2} + S_{N/2}$ ($S_{N/2}$: problem size)

$\Rightarrow O(N^2 \log N)$ (spoiled) time

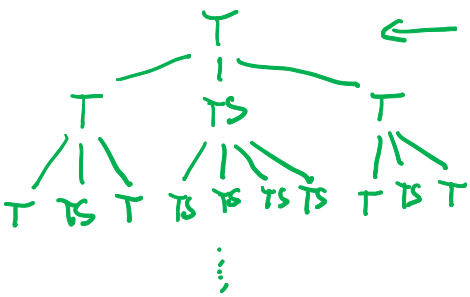
"transpose + swap" operation $TS(A,B)$:



$T_N \to 2T_{N/2} + TS_{N/2}$

$TS_N \to 4TS_{N/2}$

tree of recursions:



$\leftarrow$ level 0

height $k = \log N$ (last level)

leaves $\leq 4^{\log N} = N^2$

internal nodes $<$ # leaves $\leq N^2$

leaves: $T(a_{ii})$ do nothing, $TS(a_{ij}, a_{ji})$ swap in $O(1)$ time

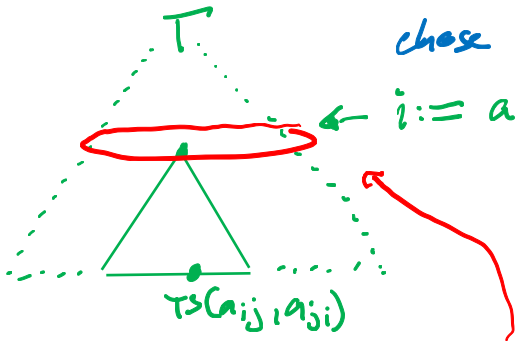
internal nodes: redistribute work (compute indices of submatrices)

$\Rightarrow O(1)$ time per node $\Rightarrow O(N^2)$ total time (as needed)

I/O analysis

Is there a "good" replacement strategy for this alg.?
↑ upper bound on OPT ↑ strategy knows B, M (but alg. does not)

memory accesses: only in TS leaves (+ recursion stack $O(\log N)$)
always cached



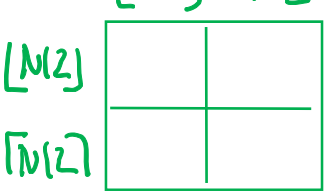
chose level $i := \arg\min_i \frac{N}{2^i} \le B$ ⇒ subproblems on this level fit into the cache
↑ subproblem size

strategy: on the level i, keep the tiles (1 for T_1, 2 for TS) in the cache for the entire subtree computation (i.e. evict cachelines of previous subtree that is not needed anymore)

⇒ equivalent to the cache-aware alg with tiles $d = \frac{N}{2^i}$

$B/2 < d \le B$ so $d = \Theta(B)$ ⇒ same $O(N^2/B+1)$ I/O complexity (assuming tall cache)

2) N is not a power of 2



⇒ all submatrices will be square or almost square (i.e. $|\#\text{rows} - \#\text{cols}| \le 1$)

⇒ the same asymptotics

Notes on cache-oblivious vs cache-aware:

- same asymptotics $O(N^2)$ time and $O(N^2/B+1)$ block transfers
- "optimal" tiles $d = B$ in cache-aware, asymptotically optimal tiles $d = \Theta(B)$ in cache-oblivious
- overhead in cache-oblivious can be reduced by stopping at some level (if we have some min. cache)

Model vs reality

I/O complexity in cache models defined w.r.t. optimal strategy.

GPT: "evict the cache line that is needed the latest"

↑ (offline, needs to know future accesses)

online strategy – knows only the current access + state of cache

Def: cost of a strategy (denoted by T_{STR} for strategy STR) w.r.t. access sequence $a_1, \dots, a_n$ = # cache misses (no assumption on initial state of cache).

A strategy STR is k-competitive for some const. $k \geq 1$ if for any cache size C and any access sequence: $T_{STR} \leq k\, T_{OPT}$.

LRU (least recently used) strategy: "evict the cache line that is the longest unused"

↑ no accesses

(implemented by priority queue)

Is LRU k-competitive for some k? No! Take C, ε s.t. $(1-\varepsilon)C > k$.

Thm: For every cache size C and $\varepsilon > 0$ there is access sequence s.th.

$$T_{LRU} \geq (1-\varepsilon)\cdot C \cdot T_{OPT}.$$

Pf: access sequence $1, \dots, C+1$ repeated large enough times to mitigate the first epoch.

epoch of size C: $1, \dots, C, C+1, 1, \dots, C-1, C, C+1, \dots$

LRU evicts: $\_\_\_, 1, 2, \dots$ ← in every access after first C accesses

EPOCH evicts: all C $\_\_\_$ $C-1$ ← once per each epoch except the first

EPOCH strategy: "evict the block not in the current epoch"

$\Rightarrow T_{LRU} \geq C \cdot T_{EPOCH}$ (except the first epoch) □