

Low Depth Cache-Oblivious Algorithms

G. E. Blelloch, P. B. Gibbons, H. V. Simhadri (SPAA 2010)

Presentation by Andrii Smutchak

October 1 2026

- **Parallel:** nested fork–join with *binary* forking $\Rightarrow$ series–parallel DAG. Costs: work W , depth D .
- **Cache-oblivious:** ideal cache of size M , line size B , both unknown to the algorithm. Cost: $Q(n; M, B)$ = number of lines transferred.
- **Glue:** Q is measured on the *sequential* execution — each forked call runs to completion before the next one starts.
- **Tall cache:** $M = \Omega(B^2)$. Every bound in this talk assumes it.

Motivation: why depth matters

Sequential Q transfers to p processors, paying only through D :

- Private caches, work stealing:

$$Q_p(n; M, B) < Q(n; M, B) + O\left(\frac{pMD}{B}\right) \quad \text{w.p. } 1 - \delta.$$

Motivation: why depth matters

Sequential Q transfers to p processors, paying only through D :

- Private caches, work stealing:

$$Q_p(n; M, B) < Q(n; M, B) + O\left(\frac{pMD}{B}\right) \quad \text{w.p. } 1 - \delta.$$

- Shared cache, PDF scheduler:

$$Q_p(n; M + pBD, B) \leq Q(n; M, B).$$

Motivation: why depth matters

Sequential Q transfers to p processors, paying only through D :

- Private caches, work stealing:

$$Q_p(n; M, B) < Q(n; M, B) + O\left(\frac{pMD}{B}\right) \quad \text{w.p. } 1 - \delta.$$

- Shared cache, PDF scheduler:

$$Q_p(n; M + pBD, B) \leq Q(n; M, B).$$

D is the only algorithm-specific overhead term
 $\Rightarrow$ design algorithms with low Q and low D .

The pMD/B term is tight (Thm 5.2), so in this analysis depth cannot be avoided.

B-Transpose: the problem

- **Input:** $h = \sqrt{n}$ rows $S_1, \dots, S_h$ (later: sorted subarrays), h buckets $\mathcal{B}_1, \dots, \mathcal{B}_h$, and an $h \times h$ matrix T with

$$T_{ij} = \langle a_{ij}, b_{ij}, \ell_{ij} \rangle : \quad S_i[a_{ij}, a_{ij} + \ell_{ij}) \longrightarrow \mathcal{B}_j[b_{ij}, b_{ij} + \ell_{ij}).$$

- **Generalized transpose:** ordinary transpose is $T_{ij} = \langle j, i, 1 \rangle$.
- **Why naive fails:** going row by row, consecutive segments land in arbitrary buckets $\Rightarrow$ up to one miss per segment: $\Theta(n)$ misses instead of n/B .

We need this to distribute keys into buckets in sorting.

B-Transpose: the algorithm

procedure $BT(i_s, i_b, k)$ $\triangleright k \times k$ block

if $k = 1$ **then**

 copy segment $T_{i_s i_b}$ (parallel loop)

else

in parallel:

$BT(i_s, i_b, k/2)$

$BT(i_s, i_b + k/2, k/2)$

$BT(i_s + k/2, i_b, k/2)$

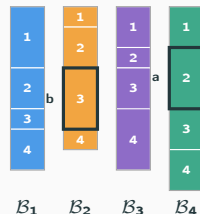
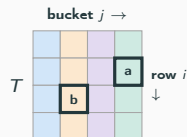
$BT(i_s + k/2, i_b + k/2, k/2)$

end if

end procedure

Recursive 4-way split on quadrants of T ;
each leaf copies one segment.

$$W = O(n), \quad D = O(\log n).$$



B-Transpose: the algorithm

procedure $\text{BT}(i_s, i_b, k)$ $\triangleright k \times k$ block

if $k = 1$ **then**

 copy segment $T_{i_s i_b}$ (parallel loop)

else

in parallel:

$\text{BT}(i_s, i_b, k/2)$

$\text{BT}(i_s, i_b + k/2, k/2)$

$\text{BT}(i_s + k/2, i_b, k/2)$

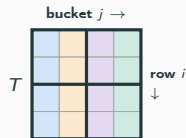
$\text{BT}(i_s + k/2, i_b + k/2, k/2)$

end if

end procedure

Recursive 4-way split on quadrants of T ;
each leaf copies one segment.

$$W = O(n), \quad D = O(\log n).$$



four independent sub-calls on the quadrants of T , run in parallel

B-Transpose: the algorithm

procedure $\text{BT}(i_s, i_b, k)$ $\triangleright k \times k$ block

if $k = 1$ **then**

copy segment $T_{i_s i_b}$ (parallel loop)

else

in parallel:

$\text{BT}(i_s, i_b, k/2)$

$\text{BT}(i_s, i_b + k/2, k/2)$

$\text{BT}(i_s + k/2, i_b, k/2)$

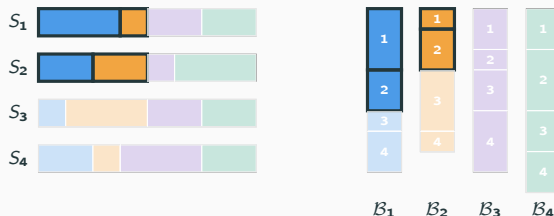
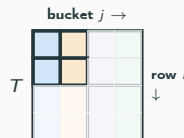
$\text{BT}(i_s + k/2, i_b + k/2, k/2)$

end if

end procedure

Recursive 4-way split on quadrants of T ;
each leaf copies one segment.

$$W = O(n), \quad D = O(\log n).$$



sub-call on a $k \times k$ block touches only k rows and k buckets

B-Transpose: the algorithm

procedure $\text{BT}(i_s, i_b, k)$ $\triangleright k \times k$ block

if $k = 1$ **then**

copy segment $T_{i_s i_b}$ (parallel loop)

else

in parallel:

$\text{BT}(i_s, i_b, k/2)$

$\text{BT}(i_s, i_b + k/2, k/2)$

$\text{BT}(i_s + k/2, i_b, k/2)$

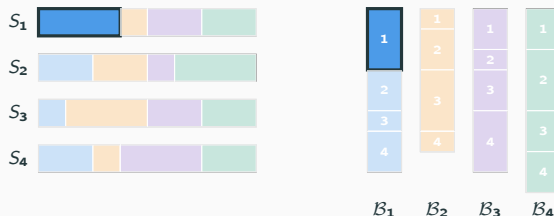
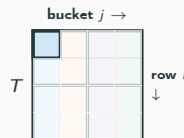
$\text{BT}(i_s + k/2, i_b + k/2, k/2)$

end if

end procedure

Recursive 4-way split on quadrants of T ;
each leaf copies one segment.

$$W = O(n), \quad D = O(\log n).$$

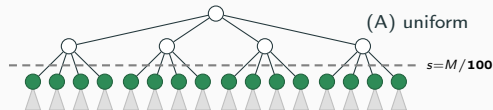


leaf ($k = 1$): **copy** $S_1[a_{11}, a_{11} + \ell_{11}) \rightarrow B_1[b_{11}, b_{11} + \ell_{11})$

B-Transpose: analysis (Lemma 2.1)

Recursion node v : size $s(v)$ = entries of T it covers, weight $w(v)$ = keys it moves.

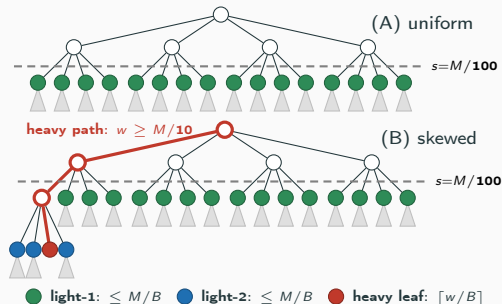
- **Light-1:** $s < M/100$, $w < M/10$, parent's size $\geq M/100$. Fits in cache (tall cache): $\leq M/B$ each, $\leq 400n/M$ of them $\Rightarrow O(n/B)$.



B-Transpose: analysis (Lemma 2.1)

Recursion node v : size $s(v)$ = entries of T it covers, weight $w(v)$ = keys it moves.

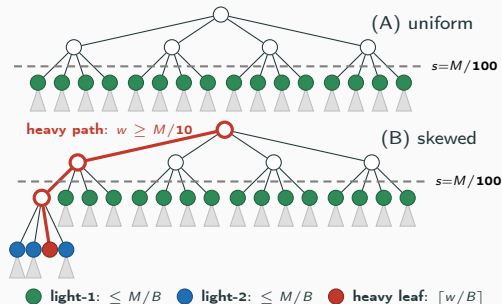
- **Light-1:** $s < M/100$, $w < M/10$, parent's size $\geq M/100$. Fits in cache (tall cache): $\leq M/B$ each, $\leq 400n/M$ of them $\Rightarrow O(n/B)$.
- **Light-2:** same, parent's weight $\geq M/10$. Cost $O(\sqrt{s} + s/B + w/B)$; $\leq 10n/M$ heavy parents per level, s drops $4\times$ per level $\Rightarrow O(n/B)$.
- **Heavy leaf:** $w \geq M/10$; $\leq W_h/(M/10)$ of them (W_h = total weight) $\Rightarrow O(W_h/B)$.



B-Transpose: analysis (Lemma 2.1)

Recursion node v : size $s(v)$ = entries of T it covers, weight $w(v)$ = keys it moves.

- **Light-1:** $s < M/100$, $w < M/10$, parent's size $\geq M/100$. Fits in cache (tall cache): $\leq M/B$ each, $\leq 400n/M$ of them $\Rightarrow O(n/B)$.
- **Light-2:** same, parent's weight $\geq M/10$. Cost $O(\sqrt{s} + s/B + w/B)$; $\leq 10n/M$ heavy parents per level, s drops $4\times$ per level $\Rightarrow O(n/B)$.
- **Heavy leaf:** $w \geq M/10$; $\leq W_h/(M/10)$ of them (W_h = total weight) $\Rightarrow O(W_h/B)$.



Classes cover all leaves $\Rightarrow Q = O(n/B)$, independent of bucket sizes.

- Optimal cache-oblivious sorts: funnelsort, distribution sort, and later variants.

$$Q = O\left(\frac{n}{B} \log_M n\right), \quad \text{but} \quad D = \Omega(\sqrt{n}).$$

- Parallel cache-oblivious analyses for specific algorithm classes: regular divide-and-conquer, dynamic programming, Gaussian elimination.

$$\text{Goal: } Q = O\left(\frac{n}{B} \log_M n\right), \quad W = O(n \log n), \quad D = \text{polylog}(n).$$

Building blocks and the merge

- **Transpose, prefix sums** (tree stored in infix order): $D = O(\log n)$, $Q = O(n/B)$.
- **Merge** (Alg. 1): find $n^{1/3}$ pivot pairs at ranks $k \left\lceil n^{2/3} \right\rceil$ by dual binary search, then merge the $n^{1/3}$ pieces in parallel.

$$D(n) \leq \log n + D(n^{2/3}) = O(\log n),$$

$$Q(n) \leq k_1 n^{1/3} (\log \frac{n}{B} + Q(n^{2/3})) \quad (n > cM) \Rightarrow Q(n) = O\left(\frac{n}{B} + n^{1/3} \log \frac{n}{B}\right) = O(n/B).$$

- **Mergesort** on this merge: $D = O(\log^2 n)$, but $Q = O\left(\frac{n}{B} \log_2 \frac{n}{M}\right)$.

$\log_2$ instead of $\log_M$ is the gap $\Rightarrow \sqrt{n}$ -way sample sort.

1. Split A into $h = \lceil \sqrt{n} \rceil$ subarrays of size $\sqrt{n}$; sort each recursively $\rightarrow S_1, \dots, S_h$.
2. Pick h pivots P .
3. SPLIT each S_i against P using merge $\rightarrow$ start offset a_{ij} and length ℓ_{ij} per bucket.
4. Build T : lengths $L = (\ell_{ij}) \rightarrow$ transpose $\rightarrow$ prefix sum per bucket $\rightarrow$ transpose back $\rightarrow$ offsets b_{ij} ; $T_{ij} = \langle a_{ij}, b_{ij}, \ell_{ij} \rangle$.
5. Run **B-Transpose**.
6. Recursively sort each bucket $\mathcal{B}_j$; concatenate.

All steps except the recursion: $O(n)$ work, $O(n/B)$ misses.

Deterministic pivots and analysis (Thm 2.2)

Sampling

- Every $\log n$ -th element of each S_i : $n / \log n$ samples.
- Mergesort the sample: $\log n$ factor smaller $\Rightarrow O(n/B)$ misses, depth $O(\log^2 n)$.
- h evenly spaced samples as pivots: every bucket has $n_j \leq 2\sqrt{n} \log n$ keys.

Deterministic pivots and analysis (Thm 2.2)

Sampling

- Every $\log n$ -th element of each S_i : $n / \log n$ samples.
- Mergesort the sample: $\log n$ factor smaller $\Rightarrow O(n/B)$ misses, depth $O(\log^2 n)$.
- h evenly spaced samples as pivots: every bucket has $n_j \leq 2\sqrt{n} \log n$ keys.

Recurrences

$$W(n) = O(n) + \sqrt{n} W(\sqrt{n}) + \sum_j W(n_j)$$

$$Q(n) = O(\lceil n/B \rceil) + \sqrt{n} Q(\sqrt{n}) + \sum_j Q(n_j)$$

$$D(n) = D(\sqrt{n}) + O(\log^2 n) + \max_j D(n_j)$$

base case $Q(n) = O(n/B)$ for $n < cM$.

Deterministic pivots and analysis (Thm 2.2)

Sampling

- Every $\log n$ -th element of each S_i : $n / \log n$ samples.
- Mergesort the sample: $\log n$ factor smaller $\Rightarrow O(n/B)$ misses, depth $O(\log^2 n)$.
- h evenly spaced samples as pivots: every bucket has $n_j \leq 2\sqrt{n} \log n$ keys.

Recurrences

$$W(n) = O(n) + \sqrt{n} W(\sqrt{n}) + \sum_j W(n_j)$$

$$Q(n) = O(\lceil n/B \rceil) + \sqrt{n} Q(\sqrt{n}) + \sum_j Q(n_j)$$

$$D(n) = D(\sqrt{n}) + O(\log^2 n) + \max_j D(n_j)$$

base case $Q(n) = O(n/B)$ for $n < cM$.

Intuition

- Each level *doubles* the total data (subarrays and buckets are both sorted).
- $\log_2 \log_M n + O(1)$ levels until pieces fit in cache
 $\Rightarrow Q = \frac{n}{B} 2^{\log_2 \log_M n + O(1)} = O\left(\frac{n}{B} \log_M n\right)$
(the $\log n$ slack in bucket sizes adds $O(1)$ levels when $M \geq \text{polylog } n$).
- Same argument down to constant size:
 $W = O(n \log n)$.
- Depth: two recursive phases on size $n^{1/2+o(1)}$; $\log^2$ shrinks by ≈ 4 per level
 $\Rightarrow D = O(\log^2 n)$.

Randomized version (Thm 2.3)

- Pick h random pivots; sort them by brute force (compare all $h^2 = n$ pairs, prefix-sum the ranks): $O(n)$ work, $O(\log n)$ depth, $O(n/B)$ misses.
- Repeat the pivot choice if some bucket exceeds $3\sqrt{n} \log n$ (probability $\leq 1/n$).
- Recursion stops at size 2^{40} .

Randomized version (Thm 2.3)

- Pick h random pivots; sort them by brute force (compare all $h^2 = n$ pairs, prefix-sum the ranks): $O(n)$ work, $O(\log n)$ depth, $O(n/B)$ misses.
- Repeat the pivot choice if some bucket exceeds $3\sqrt{n} \log n$ (probability $\leq 1/n$).
- Recursion stops at size 2^{40} .

$$\text{W.p. } > 1 - 1/n: \quad Q = O\left(\frac{n}{B} \log_M n\right), \quad W = O(n \log n), \quad D = O(\log^{1.5} n).$$

Randomized version (Thm 2.3)

- Pick h random pivots; sort them by brute force (compare all $h^2 = n$ pairs, prefix-sum the ranks): $O(n)$ work, $O(\log n)$ depth, $O(n/B)$ misses.
- Repeat the pivot choice if some bucket exceeds $3\sqrt{n} \log n$ (probability $\leq 1/n$).
- Recursion stops at size 2^{40} .

$$\text{W.p. } > 1 - 1/n: \quad Q = O\left(\frac{n}{B} \log_M n\right), \quad W = O(n \log n), \quad D = O(\log^{1.5} n).$$

- The $\log^2 n$ mergesort term disappears.
- The extra $\sqrt{\log n}$ comes from bounding retries along a path: Chernoff per level, union bound over all paths.

- Pointer chasing costs one miss per step. Fix: implement every round as a sort.

- Pointer chasing costs one miss per step. Fix: implement every round as a sort.
- **(i) Shrink the list to $n/\log n$:** remove independent sets (random mate, or deterministic coin tossing). Sizes drop geometrically $\Rightarrow$ total $O(Q_{\text{sort}}(n))$.

- Pointer chasing costs one miss per step. Fix: implement every round as a sort.
- **(i) Shrink the list to $n/\log n$:** remove independent sets (random mate, or deterministic coin tossing). Sizes drop geometrically $\Rightarrow$ total $O(Q_{\text{sort}}(n))$.
- **(ii) Pointer jumping** on the short list: $O(\log n)$ rounds, each a sort.

- Pointer chasing costs one miss per step. Fix: implement every round as a sort.
- **(i) Shrink the list to $n/\log n$:** remove independent sets (random mate, or deterministic coin tossing). Sizes drop geometrically $\Rightarrow$ total $O(Q_{\text{sort}}(n))$.
- **(ii) Pointer jumping** on the short list: $O(\log n)$ rounds, each a sort.

Thm 3.1: $Q = O(Q_{\text{sort}}(n))$, $W = O(n \log n)$, $D = O(D_{\text{sort}}(n) \log n)$.

Graph algorithms (bounded degree)

Problem	Depth	Cache complexity
List ranking	$D_{LR} = O(D_{\text{sort}} \log n)$	$O(Q_{\text{sort}}(n))$
Euler tour	$O(D_{LR})$	$O(Q_{\text{sort}}(n))$
Tree contr.	$O(D_{LR} \log n)$	$O(Q_{\text{sort}}(n))$
LCA (k q.)	$O(D_{LR})$	$O(\lceil k/n \rceil Q_{\text{sort}}(n))$
CC	$O(D_{LR} \log n)$	$O(Q_{\text{sort}}(E) \log \frac{ V }{\sqrt{M}})$
MSF	$O(D_{LR} \log n)$	$O(Q_{\text{sort}}(E) \log \frac{ V }{\sqrt{M}})$

- **Euler tour:** successor function + list ranking.
- **Tree contraction:** Euler tour + independent set.
- **LCA:** reduce to range-minima queries.
- **CC / MSF:** tree contraction.

Remark: $\log(|V|/\sqrt{M})$ vs. $\log(|V|/M)$ in the external-memory version: cache-obliviously we must wait until the *edges* fit in cache, not just the vertices.

SpMV: setting

$y = Ax$, A is $n \times n$ with $m \geq n$ nonzeros.

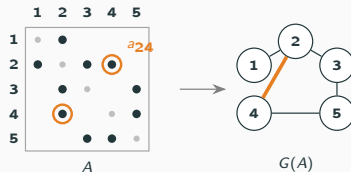
- General A : lower bound matches sorting.
- Well-shaped meshes: $O(m/B + n/M^{1/d})$; good edge separators.
- **New**: graphs with good *vertex* separators (planar, bounded genus).

n^ϵ -vertex separator theorem: $V = V_a \cup V_s \cup V_b$ with $|V_s| \leq \beta n^\epsilon$, $|V_a|, |V_b| \leq \alpha n$, no V_a - V_b edges.

Assumed: symmetric structure; building the layout is amortized, not counted.

Matrix as graph

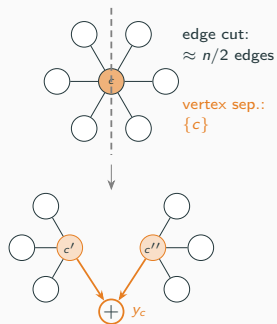
- Vertex per row/column, edge (i, j) per $a_{ij} \neq 0$: $y_i = \sum_j a_{ij}x_j$.
- Miss $\Leftarrow$ endpoints far apart in memory: a *graph layout* problem.
- Separators give locality: recurse until a piece fits in cache; extra cost only on boundaries of big pieces.



Why vertex separators?

Extra misses come from piece *boundaries*. Which boundary?

- **Edge sep.:** cut edges. Star (planar): $\approx n/2$ of them.
- **Vertex sep.:** separator vertices. Star: just $\{c\}$.
- **Trick:** *duplicate* separator vertices into both halves; add the copies on the way up (" $+$ " = node of the *vertex tree*).

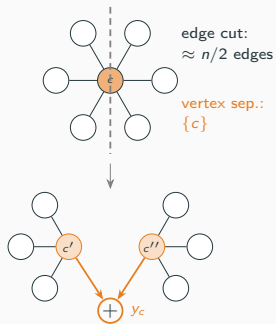


Why vertex separators?

Extra misses come from piece *boundaries*. Which boundary?

- **Edge sep.:** cut edges. Star (planar): $\approx n/2$ of them.
- **Vertex sep.:** separator vertices. Star: just $\{c\}$.
- **Trick:** *duplicate* separator vertices into both halves; add the copies on the way up (" $+$ " = node of the *vertex tree*).

Fits the recipe: halves share nothing, so they are *cache-local* (low Q) and *independent* (parallel, low D). Price: βn^ϵ additions per separator.

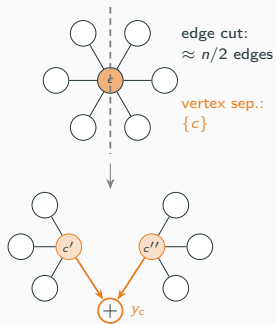


Why vertex separators?

Extra misses come from piece *boundaries*. Which boundary?

- **Edge sep.:** cut edges. Star (planar): $\approx n/2$ of them.
- **Vertex sep.:** separator vertices. Star: just $\{c\}$.
- **Trick:** *duplicate* separator vertices into both halves; add the copies on the way up (" $+$ " = node of the *vertex tree*).

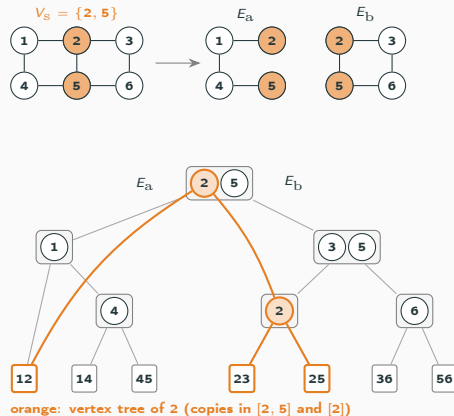
Fits the recipe: halves share nothing, so they are *cache-local* (low Q) and *independent* (parallel, low D). Price: βn^ε additions per separator.



Edges *partitioned* ($W = O(m)$), vertices *replicated*. BuildTree (next) applies this recursively.

Separator tree and layout

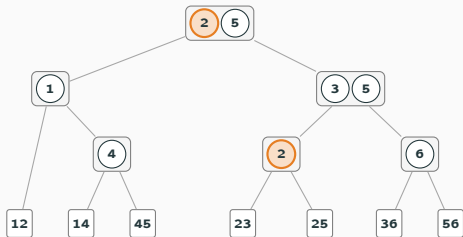
- **BuildTree:** split the *edge* set; separator vertices go to *both* sides. Leaves are single edges $\Rightarrow$ separator tree $\mathcal{T}$.
- **Vertex tree:** vertex of degree Δ has Δ leaves and $\Delta - 1$ internal copies; the top one is its root.
- **Layout:**
 - nodes of $\mathcal{T}$ in preorder, each node's vertex copies contiguous;
 - x, y reordered by preorder position of each vertex's root;
 - pointer array R to the roots.



```
procedure SPMxV( $x, \mathcal{T}$ )  
  if  $\mathcal{T}$  is a leaf, edge  $(u, v)$  then  
     $u.val \leftarrow x[v] \cdot a_{uv}$   
     $v.val \leftarrow x[u] \cdot a_{vu}$   
  else  
    in parallel: SPMxV( $x, \mathcal{T}.left$ ),  
                 SPMxV( $x, \mathcal{T}.right$ )  
    parallel for copy  $c \in \mathcal{T}.vertices$ :  
       $c.val \leftarrow c.left.val + c.right.val$   
  end if  
end procedure
```

Result y_v = value at the root of v 's vertex tree (via R).

$W = O(m)$; $D = O(\log^2 n)$: tree depth $O(\log n)$,
each parallel loop $O(\log n)$.



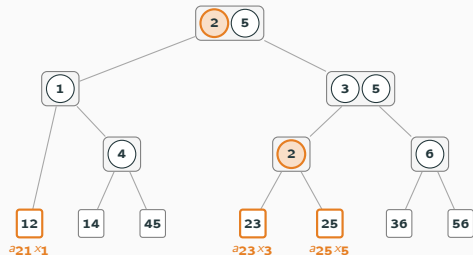
```

procedure SPMxV( $x, \mathcal{T}$ )
  if  $\mathcal{T}$  is a leaf, edge  $(u, v)$  then
     $u.val \leftarrow x[v] \cdot a_{uv}$ 
     $v.val \leftarrow x[u] \cdot a_{vu}$ 
  else
    in parallel: SPMxV( $x, \mathcal{T}.left$ ),
                  SPMxV( $x, \mathcal{T}.right$ )
    parallel for copy  $c \in \mathcal{T}.vertices$ :
       $c.val \leftarrow c.left.val + c.right.val$ 
    end if
  end procedure

```

Result y_v = value at the root of v 's vertex tree (via R).

$W = O(m)$; $D = O(\log^2 n)$: tree depth $O(\log n)$,
each parallel loop $O(\log n)$.

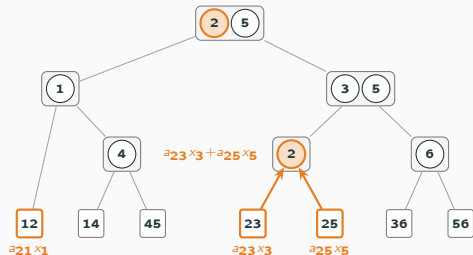


SparseMxV

```
procedure SPMxV( $x, \mathcal{T}$ )  
  if  $\mathcal{T}$  is a leaf, edge  $(u, v)$  then  
     $u.val \leftarrow x[v] \cdot a_{uv}$   
     $v.val \leftarrow x[u] \cdot a_{vu}$   
  else  
    in parallel: SPMxV( $x, \mathcal{T}.left$ ),  
                 SPMxV( $x, \mathcal{T}.right$ )  
    parallel for copy  $c \in \mathcal{T}.vertices$ :  
       $c.val \leftarrow c.left.val + c.right.val$   
    end if  
  end procedure
```

Result y_v = value at the root of v 's vertex tree (via R).

$W = O(m)$; $D = O(\log^2 n)$: tree depth $O(\log n)$,
each parallel loop $O(\log n)$.



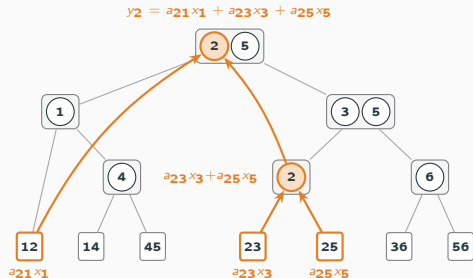
```

procedure SPMxV( $x, \mathcal{T}$ )
  if  $\mathcal{T}$  is a leaf, edge  $(u, v)$  then
     $u.val \leftarrow x[v] \cdot a_{uv}$ 
     $v.val \leftarrow x[u] \cdot a_{vu}$ 
  else
    in parallel: SPMxV( $x, \mathcal{T}.left$ ),
                  SPMxV( $x, \mathcal{T}.right$ )
    parallel for copy  $c \in \mathcal{T}.vertices$ :
       $c.val \leftarrow c.left.val + c.right.val$ 
    end if
  end procedure

```

Result y_v = value at the root of v 's vertex tree (via R).

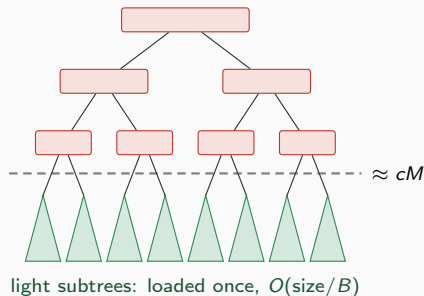
$W = O(m)$; $D = O(\log^2 n)$: tree depth $O(\log n)$,
each parallel loop $O(\log n)$.



SpMV analysis (Thm 4.1)

Heavy copy: copy in a separator node whose subtree needs $> M/k$ memory.

- Subtree size $X(n) = O(n) \Rightarrow$ subtrees on $< cM$ vertices fit in cache.



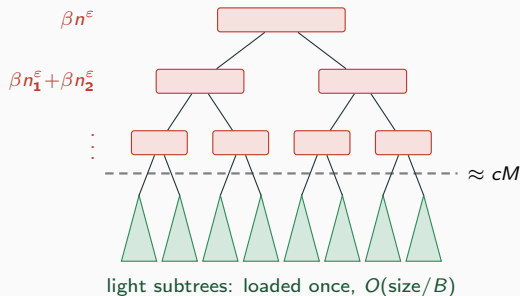
SpMV analysis (Thm 4.1)

Heavy copy: copy in a separator node whose subtree needs $> M/k$ memory.

- Subtree size $X(n) = O(n) \Rightarrow$ subtrees on $< cM$ vertices fit in cache.
- Number of heavy copies:

$$H(n) \leq H(\alpha' n) + H((1 - \alpha')n) + \beta n^\epsilon$$

for $n > cM \Rightarrow H(n) = O(n/M^{1-\epsilon})$.



SpMV analysis (Thm 4.1)

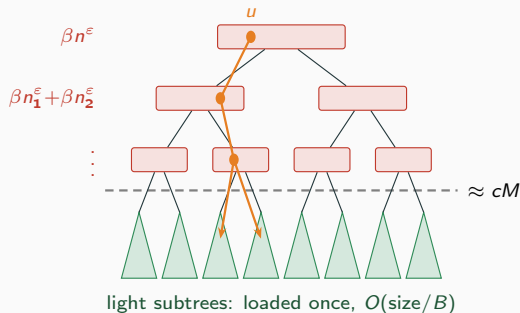
Heavy copy: copy in a separator node whose subtree needs $> M/k$ memory.

- Subtree size $X(n) = O(n) \Rightarrow$ subtrees on $< cM$ vertices fit in cache.
- Number of heavy copies:

$$H(n) \leq H(\alpha' n) + H((1 - \alpha')n) + \beta n^\epsilon$$

for $n > cM \Rightarrow H(n) = O(n/M^{1-\epsilon})$.

- Misses: light vertices $O(n/B)$; edges $O(m/B)$; each heavy copy $O(1)$.



Vertex tree of u : each green subtree that reads u is charged to one heavy copy of u .

SpMV analysis (Thm 4.1)

Heavy copy: copy in a separator node whose subtree needs $> M/k$ memory.

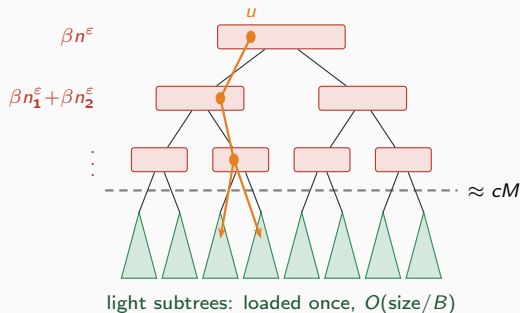
- Subtree size $X(n) = O(n) \Rightarrow$ subtrees on $< cM$ vertices fit in cache.
- Number of heavy copies:

$$H(n) \leq H(\alpha' n) + H((1 - \alpha')n) + \beta n^\varepsilon$$

for $n > cM \Rightarrow H(n) = O(n/M^{1-\varepsilon})$.

- Misses: light vertices $O(n/B)$; edges $O(m/B)$; each heavy copy $O(1)$.

$$Q = O\left(\frac{m}{B} + \frac{n}{M^{1-\varepsilon}}\right); \text{ planar } (\varepsilon = \frac{1}{2}) + \text{ tall cache: } O(m/B), \text{ optimal.}$$



Vertex tree of u : each green subtree that reads u is charged to one heavy copy of u .

SpMV: where the misses come from

Frontier: maximal subtrees of $\mathcal{T}$ on $< cM$ vertices (they fit in cache). $H = O(n/M^{1-\epsilon})$ heavy copies above it.

Data	Access pattern	Misses
Leaves (edges)	visited in preorder = memory order	$O(m/B)$
Cache-sized subtree	contiguous, run to completion, loaded once	$O(\text{size}/B + 1)$ each; $O(m/B)$ total
x, y of light vertices	contiguous range per subtree (root order)	$O(n/B)$
x of heavy vertex u	outside the subtree: 1 miss per subtree reading $u^\dagger$; $\leq h_u + 1$ such subtrees	$O(H)$ total (h_u = heavy copies of u)
Heavy copies	read two children, which may be far away	$O(1)$ each: $O(H)$
Gather y via R	scan R ; heavy roots lie above the frontier	$O(n/B + H)$
Total		$O\left(\frac{m}{B} + \frac{n}{M^{1-\epsilon}}\right)$

[†]Not justified in the paper.

Results recap

Problem	Depth	Cache complexity	
Matrix transpose ($n \times m$)	$O(\log(n + m))$	$O(\lceil nm/B \rceil)$	
Prefix sums	$O(\log n)$	$O(\lceil n/B \rceil)$	
Merge	$O(\log n)$	$O(\lceil n/B \rceil)$	
Sort (deterministic)	$O(\log^2 n)$	$O(\lceil n/B \rceil \lceil \log_M n \rceil)$	new
Sort (randomized, w.h.p.)	$O(\log^{1.5} n)$	$O(\lceil n/B \rceil \lceil \log_M n \rceil)$	new
SpMV (m nonzeros, n^ε sep.)	$O(\log^2 n)$	$O(\lceil m/B + n/M^{1-\varepsilon} \rceil)$	new

All work-optimal; cache complexities match the best sequential algorithms ($M = \Omega(B^2)$).

Bounds extend to multi-level private and shared hierarchies (§5, not covered).

Strong

- Simple, machine-independent recipe: low Q plus low D .
- First results of their kind for sorting and SpMV.
- Reusable primitive: B-Transpose.
- pMD/B overhead shown tight.

Weak

- Proofs are sketchy and some steps are wrong or missing (Lemma 2.1, Thm 2.2 depth, randomized depth, Thm 4.1); the results likely still hold.
- Large constant: base case 2^{40} (randomized).
- Edge cases: base case $n < 10$ too small ($2\sqrt{n} \log n < n$ needs $n \gtrsim 256$); equal keys can put all n keys in one bucket.
- No experiments; SpMV preprocessing not analyzed.

1. **Can we beat these bounds?** With work stealing the parallel run pays up to $O(pMD/B)$ extra misses; with PDF it needs pBD extra cache. Could a smarter scheduler (e.g., one that knows the cache sizes) or a different algorithm design reduce or remove this dependence on depth? And can sorting depth go below $\log^2 n / \log^{1.5} n$ while keeping optimal Q ?
2. **Is optimal Q worth it?** Mergesort with the new merge is already low-depth and only loses $\log_2(n/M)$ vs. $\log_M n$; for $n = 2^{30}$, $M = 2^{20}$ that's 10 vs. 1.5. Is the difference worth a much more complex algorithm?
3. **Is this practical?** Constants like 2^{40} , and no experiments. Does that matter for a theory paper? What would you want to see in an experimental follow-up?
4. **Where does SpMV apply?** It works best on planar-like graphs. Which real-world matrices have good vertex separators, and which don't (e.g., social networks)?
5. **Is ignoring SpMV preprocessing fair?** Building the separator tree is expensive and not counted. For which applications is that reasonable, and when is it not?

- **Better sorting depth:** SPMS (Cole & Ramachandran, ICALP '10; TOPC '17) — deterministic, optimal work and Q , depth $O(\log n \cdot \log \log n)$; also bounds false-sharing costs.

- **Better sorting depth:** SPMS (Cole & Ramachandran, ICALP '10; TOPC '17) — deterministic, optimal work and Q , depth $O(\log n \cdot \log \log n)$; also bounds false-sharing costs.
- **Binary-forking model:** Blelloch, Fineman, Gu, Sun (SPAA '20) formalize the model used here and give optimal-work, $O(\log n)$ -span (randomized) algorithms for sorting, list ranking, tree contraction, etc. (some relying on test-and-set; they note SPMS is still the best sort without it).

- **Better sorting depth:** SPMS (Cole & Ramachandran, ICALP '10; TOPC '17) — deterministic, optimal work and Q , depth $O(\log n \cdot \log \log n)$; also bounds false-sharing costs.
- **Binary-forking model:** Blelloch, Fineman, Gu, Sun (SPAA '20) formalize the model used here and give optimal-work, $O(\log n)$ -span (randomized) algorithms for sorting, list ranking, tree contraction, etc. (some relying on test-and-set; they note SPMS is still the best sort without it).
- **Hierarchy-aware schedulers:** space-bounded schedulers and the PCO cost model (Blelloch, Fineman, Gibbons, Simhadri, SPAA '11) handle multi-level caches and irregular tasks such as B-Transpose; evaluated experimentally in Simhadri et al. (SPAA '14; TOPC '16).

- These slides were produced with an LLM, working from my presentation plan.