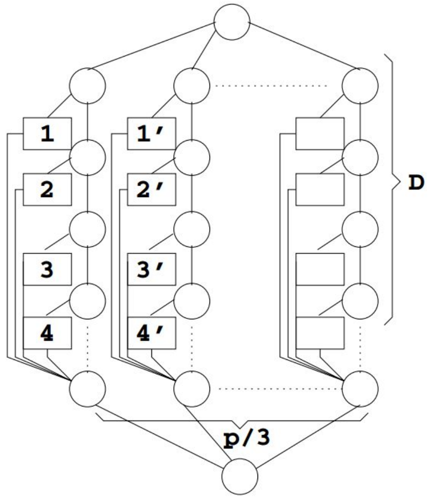
Low depth cache-oblivious algorithms

Guy E. Blelloch, Phillip B. Gibbons, Harsha Vardhan Simhadri

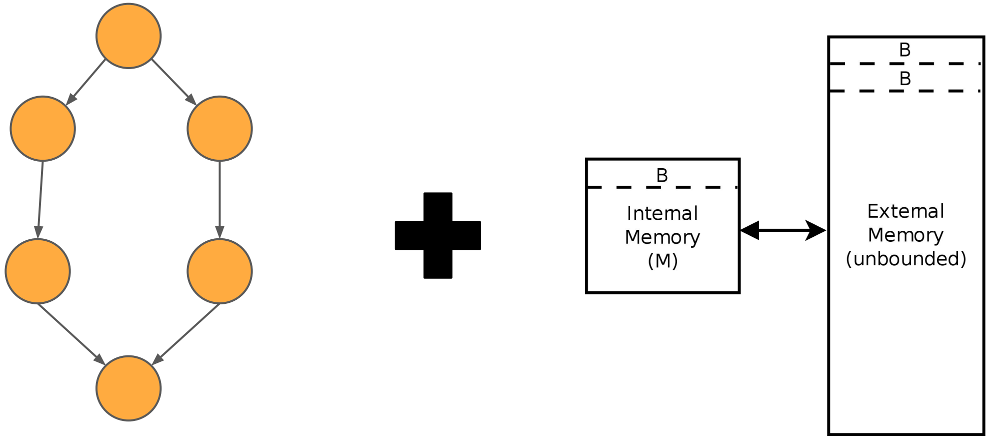


Presenter: Jingnan Shi

Outline

- Motivation
- Low-depth Cache Oblivious Sorting
    - Deterministic
    - Random
    - Applications
- Low-depth Cache Oblivious Sparse Matrix Vector Multiplication
- Mapping to Different Multi-level Memory Hierarchies
    - Parallel Multi-level Distributed Hierarchy (PMDH)
        - Bounds
    - Parallel Multi-level Shared Hierarchy (PMSH)
        - Bounds

Motivation



- Interested in studying locality of algorithms written with dynamic nested parallelism
- Combining work & depth analysis with cache complexity from sequential cache-oblivious model

Why low depth?

For a work-stealing scheduler (binary forking, single level of cache) [Acar et al., 2000]:

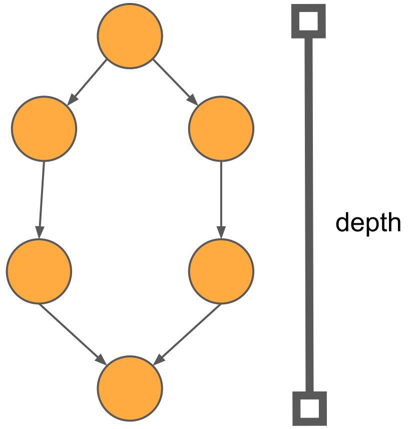
$$Q_p(n; M, B) < Q(n; M, B) + O(pMD/B) \text{ with probability } 1 - \delta$$

Parallel cache complexity

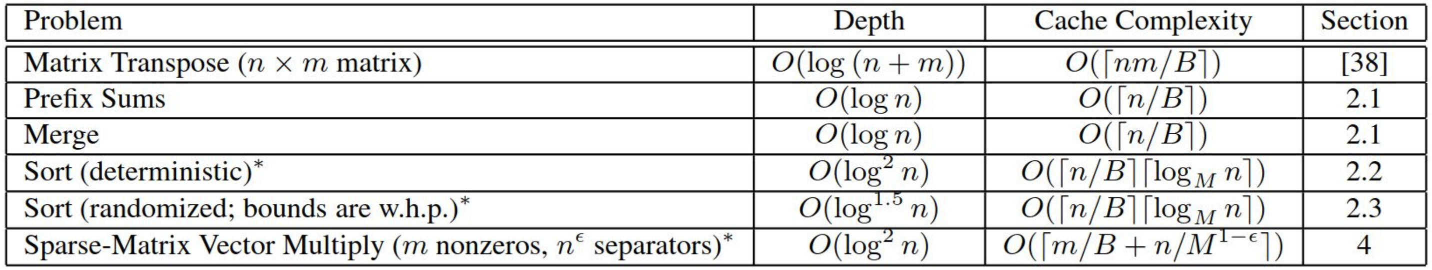
Natural sequential cache complexity

This suggests an approach for developing cache-efficient parallel algorithms:

- Develop a nested-parallel algorithm with (1) low sequential cache complexity, and (2) low depth
- Bound the results on a parallel machine



Summary of Low-depth Cache-Oblivious Algorithms



Problem	Depth	Cache Complexity	Section
Matrix Transpose ($n \times m$ matrix)	$O(\log(n+m))$	$O(\lceil nm/B \rceil)$	[38]
Prefix Sums	$O(\log n)$	$O(\lceil n/B \rceil)$	2.1
Merge	$O(\log n)$	$O(\lceil n/B \rceil)$	2.1
Sort (deterministic)*	$O(\log^2 n)$	$O(\lceil n/B \rceil \lceil \log_M n \rceil)$	2.2
Sort (randomized; bounds are w.h.p.)*	$O(\log^{1.5} n)$	$O(\lceil n/B \rceil \lceil \log_M n \rceil)$	2.3
Sparse-Matrix Vector Multiply (m nonzeros, n^ϵ separators)*	$O(\log^2 n)$	$O(\lceil m/B + n/M^{1-\epsilon} \rceil)$	4

Figure 1: Low-depth cache-oblivious algorithms. New algorithms are marked (*). All algorithms are work optimal and their cache complexities match the best sequential algorithms. The bounds assume $M = \Omega(B^2)$.

Sorting: Preliminaries

Algorithm 1 MERGE$((A, s_A, l_A), (B, s_B, l_B), (C, s_C))$

Merges $A[s_A : s_A + l_A)$ and $B[s_B : s_B + l_B)$ into array $C[s_C : s_C + l_A + l_B)$

```
1: if $l_B = 0$ then
2:     Copy $A[s_A : s_A + l_A)$ to $C[s_C : s_C + l_A)$
3: else if $l_A = 0$ then
4:     Copy $B[s_B : s_B + l_B)$ to $C[s_C : s_C + l_B)$
5: else
6:     $\forall k \in [1 : \lfloor n^{1/3} \rfloor]$, find pivots $(a_k, b_k)$ such that $a_k + b_k = k\lceil n^{2/3} \rceil$ and $A[s_A + a_k] \leq B[s_B + b_k + 1]$ and $B[s_B + b_k] \leq A[s_A + a_k + 1]$.
7:     $\forall k \in [1 : \lfloor n^{1/3} \rfloor]$, MERGE$((A, s_A + a_k, a_{k+1} - a_k), (B, s_B + b_k, b_{k+1} - b_k), (C, s_C + a_k + b_k))$
8: end if
```

Cache complexity:

$$Q(n; M, B) \leq \begin{cases} k_1 n^{1/3}\big(\log(n/B) + Q\big(n^{2/3}; M, B\big)\big) & n > cM \\ k_2 n/B + 1 & \text{otherwise} \end{cases}$$

Use tall cache assumption, and assume n > cM, we have:

$$Q(n; M, B) = O(\lceil n/B \rceil)$$

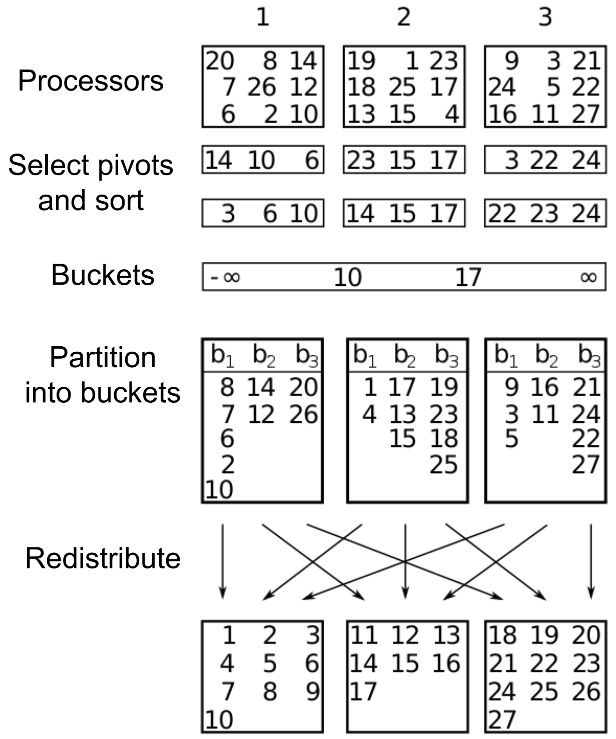
Depth:

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

Sorting: Sample Sort (Frazer and McKellar, 1970)

Pseudocode:

1. If average bucket size is smaller than a threshold, switch to quicksort
2. Sample p−1 elements from the input (the pivots).
3. Sort the pivots; each pair of adjacent pivots then defines a bucket.
4. Loop over the data, placing each element in the appropriate bucket (using binary search).
5. Sort each of the buckets.
6. Return the concatenation of all the buckets



Sorting: COSORT

```
Algorithm COSORT($A, n$)
if $n < 10$ then
    return Sort $A$ sequentially
end if
$h \leftarrow \lceil \sqrt{n} \rceil$
$\forall i \in [1:h]$, Let $A_i \leftarrow A[h(i-1)+1 : hi]$
$\forall i \in [1:h]$, $S_i \leftarrow$ COSORT($A_i, h$)
repeat
    Pick an appropriate sorted pivot set $\mathcal{P}$ of size $h$
    $\forall i \in [1:h]$, $M_i \leftarrow$ SPLIT($S_i, \mathcal{P}$)
    {Each array $M_i$ contains for each bucket $j$ a start location in $S_i$ for bucket
    $j$ and a length of how many entries are in that bucket, possibly 0.}
    $L \leftarrow h \times h$ matrix formed by rows $M_i$ with just the lengths
    $L^T \leftarrow$ TRANSPOSE($L$)
    $\forall i \in [1:h]$, $O_i \leftarrow$ PREFIX-SUM($L_i^T$)
    $O^T \leftarrow$ TRANSPOSE($O$)    {$O_i$ is the $i$th row of $O$}
    $\forall i, j \in [1:h]$, $T_{i,j} \leftarrow \langle M_{i,j}\langle 1\rangle, O_{i,j}^T, M_{i,j}\langle 2\rangle \rangle$
    {Each triple corresponds to an offset in row $i$ for bucket $j$, an offset in
    bucket $j$ for row $i$ and the length to copy.}
until No bucket is too big
Let $B_1, B_2, \ldots, B_h$ be arrays (buckets) of sizes dictated by $T$
B-TRANSPOSE($S, B, T, 1, 1, h$)
$\forall i$, $B'_i \leftarrow$ COSORT($B_i$, length($B_i$))
return $B'_1 || B'_2 || \ldots || B'_h$
```

If input array is small, use sequential sort

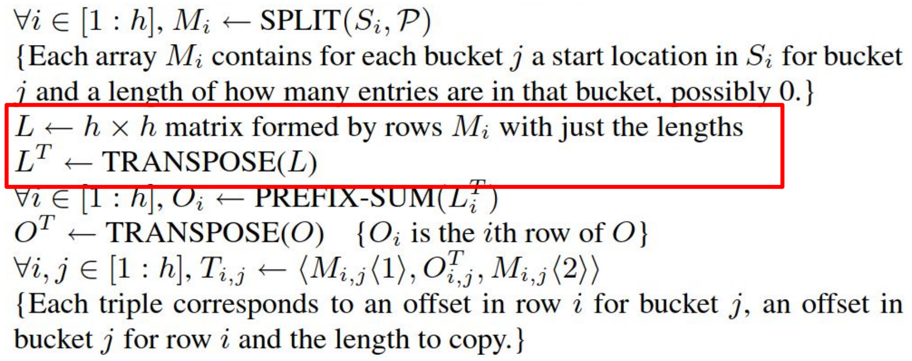
Split into sqrt(n) subarrays and recursively sort each subarray

Sample pivots (determinstic or randomized)

Split the sorted subarrays (determine in each subarray the buckets' starting offsets and length)

Sort the buckets and return the concatenated buckets

Sorting: COSORT



```
$\forall i \in [1:h]$, $M_i \leftarrow$ SPLIT($S_i, \mathcal{P}$)
{Each array $M_i$ contains for each bucket $j$ a start location in $S_i$ for bucket $j$ and a length of how many entries are in that bucket, possibly 0.}
$L \leftarrow h \times h$ matrix formed by rows $M_i$ with just the lengths
$L^T \leftarrow$ TRANSPOSE($L$)
$\forall i \in [1:h]$, $O_i \leftarrow$ PREFIX-SUM($L_i^T$)
$O^T \leftarrow$ TRANSPOSE($O$)    {$O_i$ is the $i$th row of $O$}
$\forall i, j \in [1:h]$, $T_{i,j} \leftarrow \langle M_{i,j}\langle 1\rangle, O^T_{i,j}, M_{i,j}\langle 2\rangle\rangle$
{Each triple corresponds to an offset in row $i$ for bucket $j$, an offset in bucket $j$ for row $i$ and the length to copy.}
```

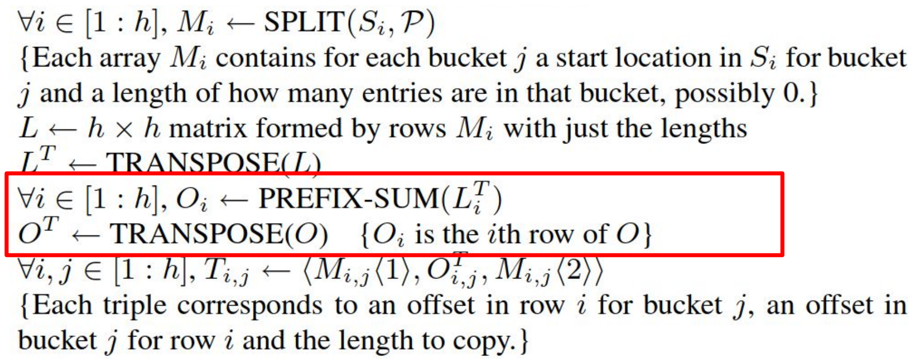
L matrix:

- h x h, where 1st dimension indicate ith subarray and 2nd dimension indicate the jth bucket
- (i,j) value: the length of the jth bucket inside the ith subarray

L matrix transpose:

- Each row represents the lengths of the ith bucket within all subarrays
- (i,j) value: the length of the ith bucket inside the jth subarray

Sorting: COSORT

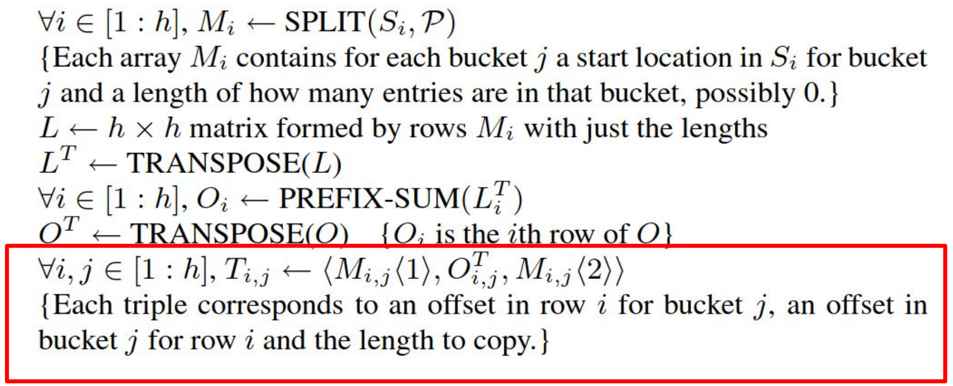


```
$\forall i \in [1:h]$, $M_i \leftarrow$ SPLIT($S_i, \mathcal{P}$)
{Each array $M_i$ contains for each bucket $j$ a start location in $S_i$ for bucket $j$ and a length of how many entries are in that bucket, possibly 0.}
$L \leftarrow h \times h$ matrix formed by rows $M_i$ with just the lengths
$L^T \leftarrow$ TRANSPOSE($L$)
$\forall i \in [1:h]$, $O_i \leftarrow$ PREFIX-SUM($L^T_i$)
$O^T \leftarrow$ TRANSPOSE($O$)    {$O_i$ is the $i$th row of $O$}
$\forall i, j \in [1:h]$, $T_{i,j} \leftarrow \langle M_{i,j}\langle 1\rangle, O^T_{i,j}, M_{i,j}\langle 2\rangle\rangle$
{Each triple corresponds to an offset in row $i$ for bucket $j$, an offset in bucket $j$ for row $i$ and the length to copy.}
```

ith row of matrix O:

- Prefix-sum of the ith row of L transpose
  - L^T(i,j): the length of the ith bucket inside the jth subarray
- Gives the starting offsets of ith bucket within each subarray

Sorting: COSORT



```
$\forall i \in [1:h]$, $M_i \leftarrow$ SPLIT$(S_i, \mathcal{P})$
{Each array $M_i$ contains for each bucket $j$ a start location in $S_i$ for bucket $j$ and a length of how many entries are in that bucket, possibly 0.}
$L \leftarrow h \times h$ matrix formed by rows $M_i$ with just the lengths
$L^T \leftarrow$ TRANSPOSE$(L)$
$\forall i \in [1:h]$, $O_i \leftarrow$ PREFIX-SUM$(L_i^T)$
$O^T \leftarrow$ TRANSPOSE$(O)$   {$O_i$ is the $i$th row of $O$}
$\forall i, j \in [1:h]$, $T_{i,j} \leftarrow \langle M_{i,j}\langle 1\rangle, O^T_{i,j}, M_{i,j}\langle 2\rangle\rangle$
{Each triple corresponds to an offset in row $i$ for bucket $j$, an offset in bucket $j$ for row $i$ and the length to copy.}
```

T_{i,j} triplet:

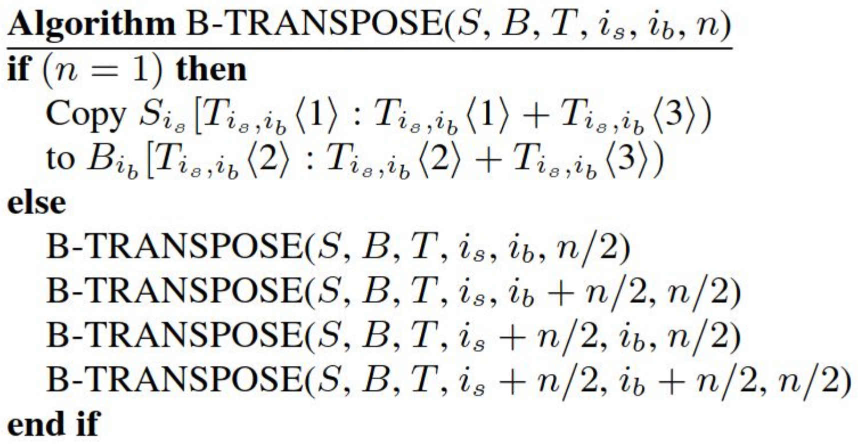
- M_{i,j}<1>: Offset of jth bucket in subarray i
- O^T_{i,j}: Offset of ith subarray in bucket j
- M_{i,j}<2>: Length to copy

Sorting: COSORT

```
**Algorithm** COSORT($A, n$)
if $n < 10$ then
    return Sort $A$ sequentially
end if
$h \leftarrow \lceil \sqrt{n} \rceil$
$\forall i \in [1:h]$, Let $A_i \leftarrow A[h(i-1)+1 : hi]$
$\forall i \in [1:h]$, $S_i \leftarrow$ COSORT($A_i$, $h$)
repeat
    Pick an appropriate sorted pivot set $\mathcal{P}$ of size $h$
    $\forall i \in [1:h]$, $M_i \leftarrow$ SPLIT($S_i, \mathcal{P}$)
    {Each array $M_i$ contains for each bucket $j$ a start location in $S_i$ for bucket
    $j$ and a length of how many entries are in that bucket, possibly 0.}
    $L \leftarrow h \times h$ matrix formed by rows $M_i$ with just the lengths
    $L^T \leftarrow$ TRANSPOSE($L$)
    $\forall i \in [1:h]$, $O_i \leftarrow$ PREFIX-SUM($L_i^T$)
    $O^T \leftarrow$ TRANSPOSE($O$)    {$O_i$ is the $i$th row of $O$}
    $\forall i, j \in [1:h]$, $T_{i,j} \leftarrow \langle M_{i,j}\langle 1 \rangle, O_{i,j}^T, M_{i,j}\langle 2 \rangle \rangle$
    {Each triple corresponds to an offset in row $i$ for bucket $j$, an offset in
    bucket $j$ for row $i$ and the length to copy.}
until No bucket is too big
Let $B_1, B_2, \ldots, B_h$ be arrays (buckets) of sizes dictated by $T$
B-TRANSPOSE($S$, $B$, $T$, 1, 1, $h$)
$\forall i$, $B_i' \leftarrow$ COSORT($B_i$, length($B_i$))
return $B_1' || B_2' || \ldots || B_h'$
```

After we have the mappings, we need to transfer the actual keys to the right buckets.

Sorting: B-Transpose



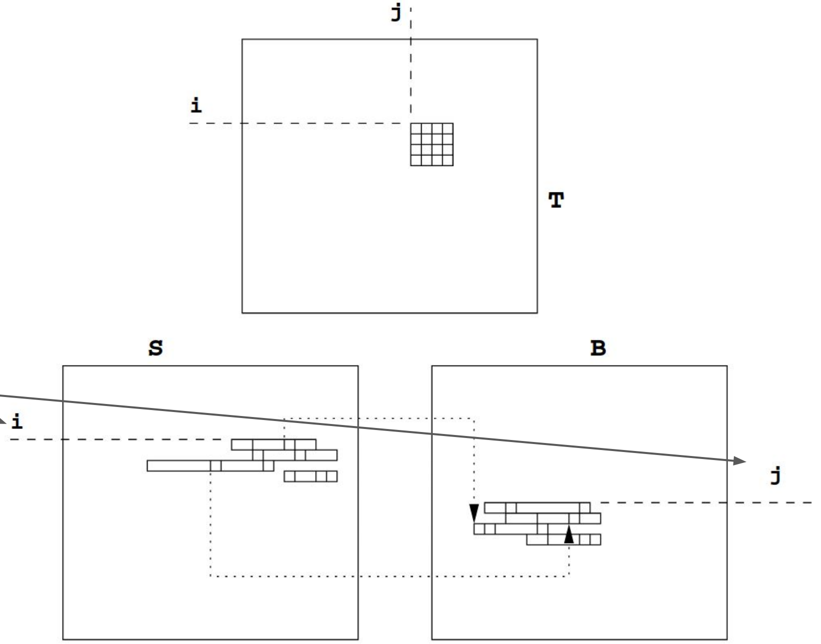
```
**Algorithm** B-TRANSPOSE($S, B, T, i_s, i_b, n$)
**if** $(n = 1)$ **then**
    Copy $S_{i_s}[T_{i_s,i_b}\langle 1\rangle : T_{i_s,i_b}\langle 1\rangle + T_{i_s,i_b}\langle 3\rangle)$
    to $B_{i_b}[T_{i_s,i_b}\langle 2\rangle : T_{i_s,i_b}\langle 2\rangle + T_{i_s,i_b}\langle 3\rangle)$
**else**
    B-TRANSPOSE($S, B, T, i_s, i_b, n/2$)
    B-TRANSPOSE($S, B, T, i_s, i_b + n/2, n/2$)
    B-TRANSPOSE($S, B, T, i_s + n/2, i_b, n/2$)
    B-TRANSPOSE($S, B, T, i_s + n/2, i_b + n/2, n/2$)
**end if**
```

- Inputs:
    - S: subarray (each row is a subarray)
    - B: buckets (each row is a bucket)
    - T: mapping matrix
    - i_s: starting subarray index
    - i_b: starting bucket index
    - n: size of the matrix
- Four-way divide and conquer
- Copy partitions of S matrix to B matrix based on the mappings provided by the T matrix

Sorting: B-Transpose

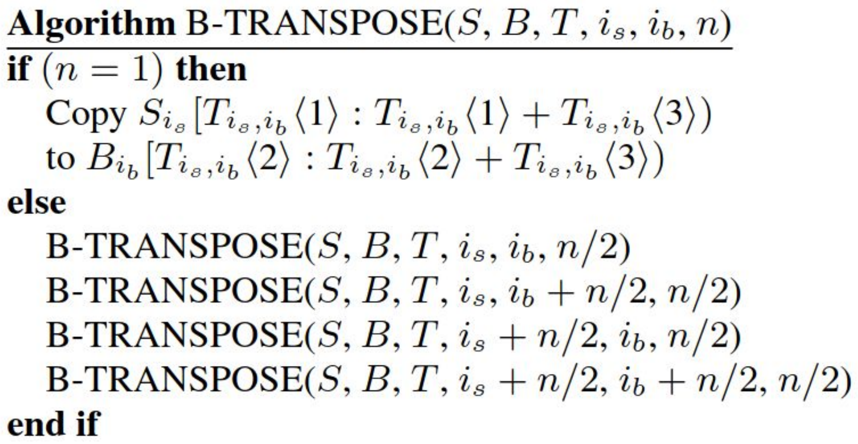
T_{i,j} triplet:

- M_{i,j}<1>: Offset of jth bucket in subarray i
- O^T_{i,j}: Offset of ith subarray in bucket j
- M_{i,j}<2>: Length to copy



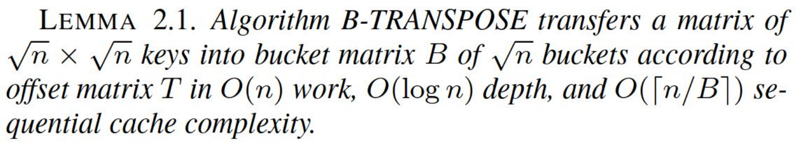
Bucket transpose diagram: The 4x4 entries shown for T dictate the mapping from the 16 depicted segments of S to the 16 depicted segments of B. Arrows highlight the mapping for two of the segments.

Sorting: B-Transpose



```
Algorithm B-TRANSPOSE($S, B, T, i_s, i_b, n$)
if ($n = 1$) then
    Copy $S_{i_s}[T_{i_s,i_b}\langle 1\rangle : T_{i_s,i_b}\langle 1\rangle + T_{i_s,i_b}\langle 3\rangle)$
    to $B_{i_b}[T_{i_s,i_b}\langle 2\rangle : T_{i_s,i_b}\langle 2\rangle + T_{i_s,i_b}\langle 3\rangle)$
else
    B-TRANSPOSE($S, B, T, i_s, i_b, n/2$)
    B-TRANSPOSE($S, B, T, i_s, i_b + n/2, n/2$)
    B-TRANSPOSE($S, B, T, i_s + n/2, i_b, n/2$)
    B-TRANSPOSE($S, B, T, i_s + n/2, i_b + n/2, n/2$)
end if
```

Bounds:



LEMMA 2.1. *Algorithm B-TRANSPOSE transfers a matrix of $\sqrt{n} \times \sqrt{n}$ keys into bucket matrix B of $\sqrt{n}$ buckets according to offset matrix T in $O(n)$ work, $O(\log n)$ depth, and $O(\lceil n/B \rceil)$ sequential cache complexity.*

Proof sketch:

- Split the recursion tree nodes into three types
- For each type of nodes, analyse the cache misses separately

Sorting: Deterministic Sampling

- choose every (logn)-th element from each of the subarrays as a sample.
- Sort the sample set with mergesort outlined above
  - Smaller than the given dataset by a factor of log(n)
  - Cache efficient - No more than O(n/B) cache misses
- Pick sqrt(b) evenly spaced samples as pivots

THEOREM 2.2. *On an input of size n, the deterministic COSORT has $Q(n; M, B) = O(\lceil n/B \rceil \lceil \log_M n \rceil)$ sequential cache complexity, $O(n \log n)$ work, and $O(\log^2 n)$ depth.*

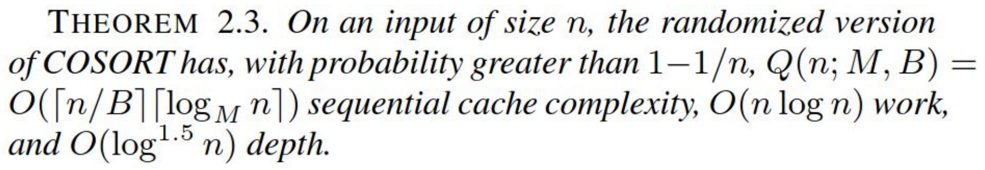
$$W(n) = O(n) + \sqrt{n}W(\sqrt{n}) + \sum_{i=1}^{\sqrt{n}} W(n_i)$$

$$D(n) = O(\log^2 n) + \max_{i=1}^{\sqrt{n}}\{D(n_i)\}$$

$$Q(n; M, B) = O\left(\left\lceil \frac{n}{B} \right\rceil\right) + \sqrt{n}Q(\sqrt{n}; M, B) + \sum_{i=1}^{\sqrt{n}} Q(n_i; M, B),$$

Sorting: Randomized Sampling

- Randomly pick sqrt(n) elements for pivots
- Brute force sort the sampled elements, and use the sorted set as pivots
- If one of the buckets is too large, the process of selecting pivots and computing bucket boundaries is repeated

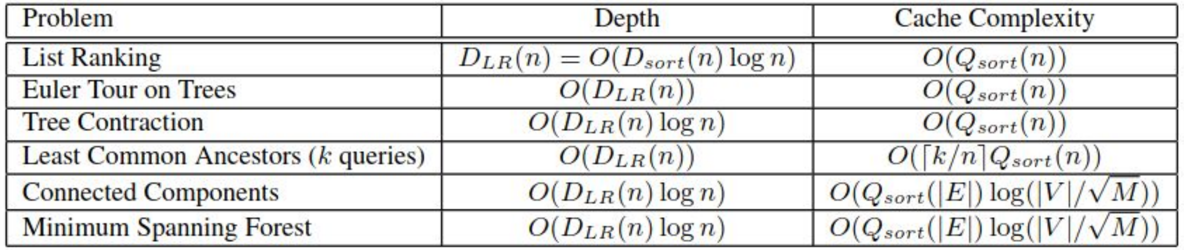


THEOREM 2.3. *On an input of size n, the randomized version of COSORT has, with probability greater than $1-1/n$, $Q(n; M, B) = O(\lceil n/B \rceil \lceil \log_M n \rceil)$ sequential cache complexity, $O(n \log n)$ work, and $O(\log^{1.5} n)$ depth.*

Proof sketch:

- Work:
  - Each iteration of the loop requires O(n) work
  - Terminates with probability 1-1/n
- Cache:
  - Incur O(n/B) cache misses with high probability
- Depth:
  - Chernoff bounds on DAG

Sorting: Applications

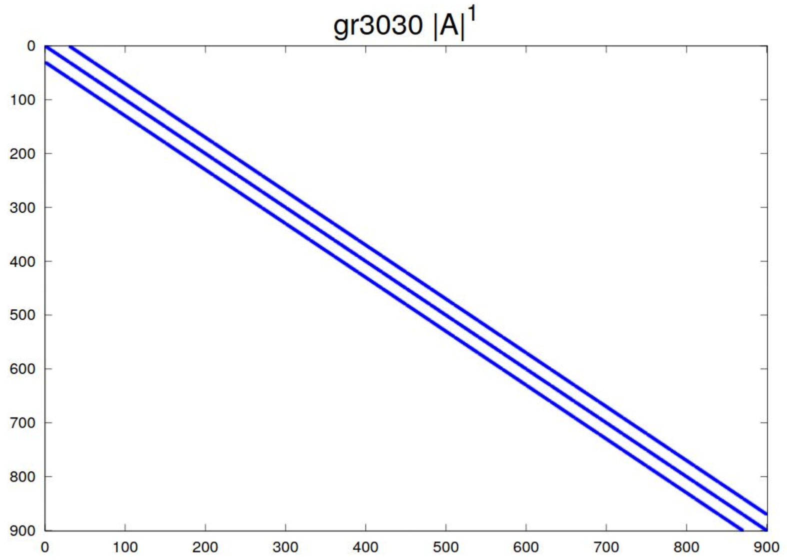


Problem	Depth	Cache Complexity
List Ranking	$D_{LR}(n) = O(D_{sort}(n) \log n)$	$O(Q_{sort}(n))$
Euler Tour on Trees	$O(D_{LR}(n))$	$O(Q_{sort}(n))$
Tree Contraction	$O(D_{LR}(n) \log n)$	$O(Q_{sort}(n))$
Least Common Ancestors (k queries)	$O(D_{LR}(n))$	$O(\lceil k/n \rceil Q_{sort}(n))$
Connected Components	$O(D_{LR}(n) \log n)$	$O(Q_{sort}(\|E\|) \log(\|V\|/\sqrt{M}))$
Minimum Spanning Forest	$O(D_{LR}(n) \log n)$	$O(Q_{sort}(\|E\|) \log(\|V\|/\sqrt{M}))$

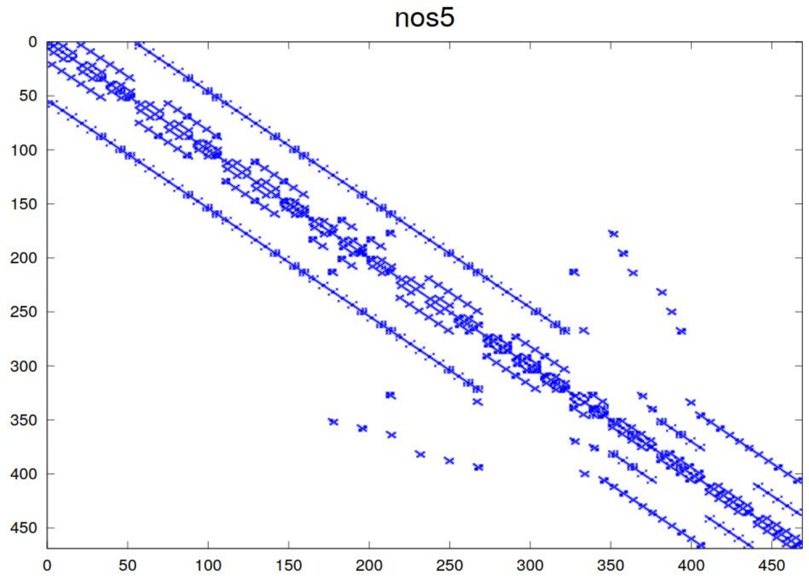
Figure 3: Low-depth cache-oblivious graph algorithms. All algorithms are deterministic. The bounds assume $M = \Omega(B^2)$. D_{sort} and Q_{sort} are the depth and cache complexity of cache-oblivious sorting.

```
Segmentation fault
Segmentation fault
Segmentation fault
```

SpMat-Vec Multiply: Intro

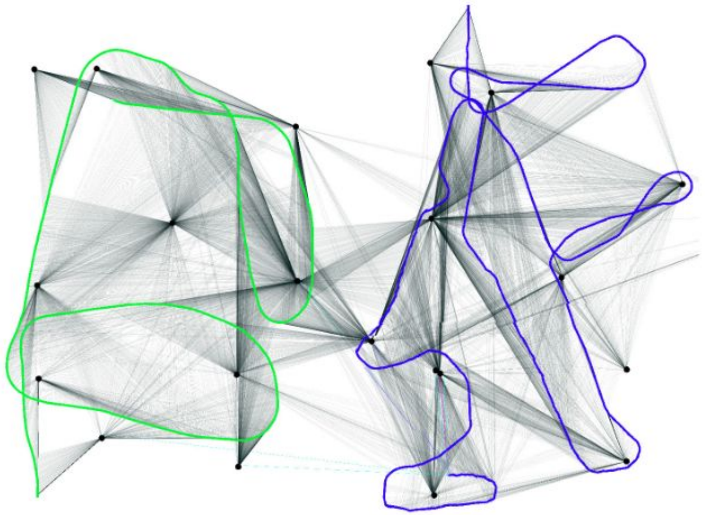


Finite-difference Laplacian (https://www.it.uu.se/education/phd_studies/phd_courses/pasc/lecture-1)



Structural Mechanics (https://www.it.uu.se/education/phd_studies/phd_courses/pasc/lecture-1)

SpMat-Vec Multiply: Intro



SLAM
(https://www.cs.cmu.edu/~kaess/pub/Dellaert17fnt.pdf)

$$\begin{bmatrix} \Lambda_{11} & & \Lambda_{13} & \Lambda_{14} & \\ & \Lambda_{22} & & & \Lambda_{25} \\ \Lambda_{31} & & \Lambda_{33} & \Lambda_{34} & \\ \Lambda_{41} & & \Lambda_{43} & \Lambda_{44} & \Lambda_{45} \\ & \Lambda_{52} & & \Lambda_{54} & \Lambda_{55} \end{bmatrix}$$

Figure 3.3: The Hessian matrix $\Lambda \triangleq A^\top A$ for the toy SLAM problem.

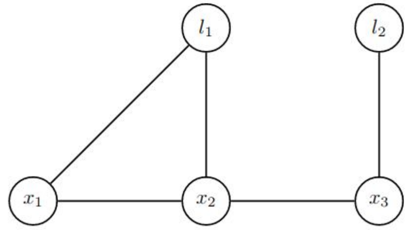
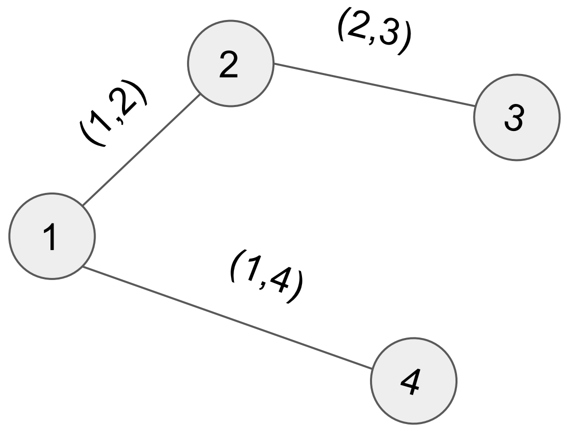


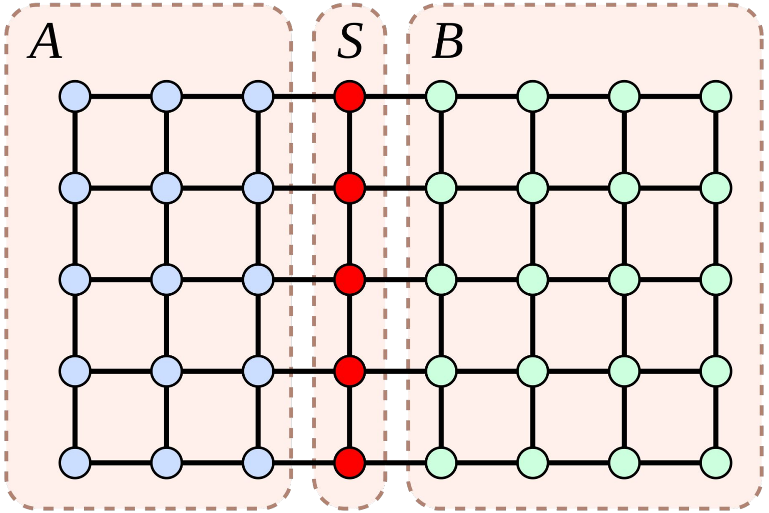
Figure 3.4: The Hessian matrix Λ can be interpreted as the matrix associated with the Markov random field representation for the problem.

SpMat-Vec Multiply: Sparse Matrix as a Graph

	1	2	3	4
1	0	1	0	1
2	1	0	1	0
3	0	1	0	0
4	1	0	0	0



SpMat-Vec Multiply: Vertex Separator



Vertex separators in a planar graph

(https://en.wikipedia.org/wiki/Planar_separator_theorem#/media/File:Grid_separator.svg)

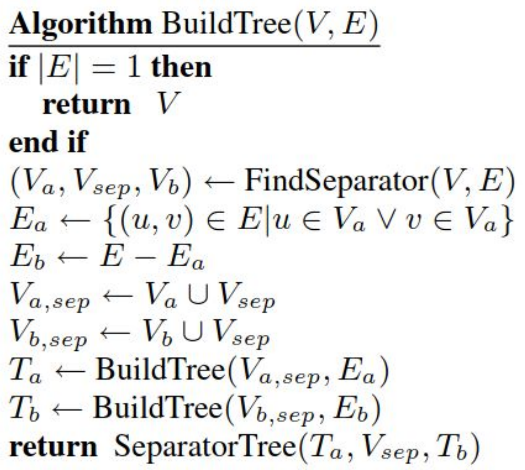
f(n)-vertex separator theorem:

- S be a class of graph closed under the subgraph relation
- If there are constants α<1 and β>0 such that every graph G(V,E) in S with n vertices can be partitioned into three sets of vertices V_a,V_s,V_b such that

$$|V_s| \leq \beta f(n), |V_a|, |V_b| \leq \alpha n$$

$$\{(u,v) \in E \mid (u \in V_a \wedge v \in V_b) \vee (u \in V_b \wedge v \in V_a)\} = \emptyset$$

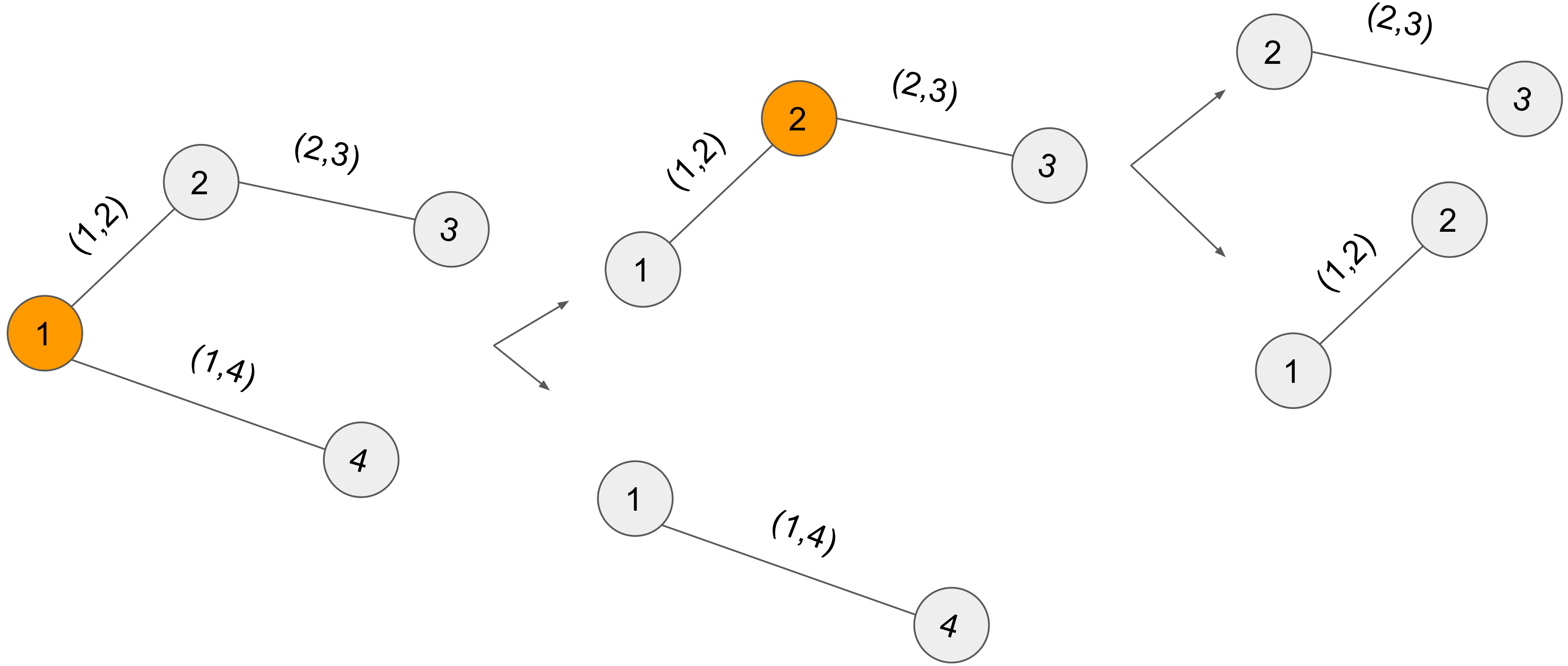
SpMat-Vec Multiply: Separator Tree



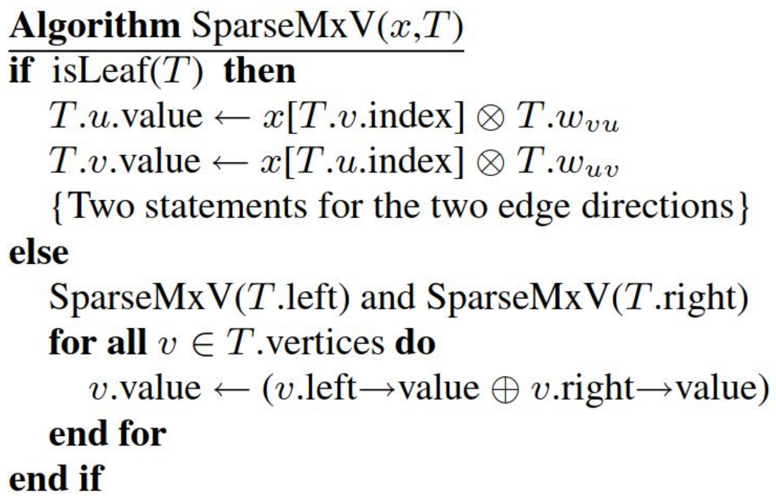
```
**Algorithm** BuildTree$(V, E)$
**if** $|E| = 1$ **then**
    **return** $V$
**end if**
$(V_a, V_{sep}, V_b) \leftarrow$ FindSeparator$(V, E)$
$E_a \leftarrow \{(u, v) \in E | u \in V_a \vee v \in V_a\}$
$E_b \leftarrow E - E_a$
$V_{a,sep} \leftarrow V_a \cup V_{sep}$
$V_{b,sep} \leftarrow V_b \cup V_{sep}$
$T_a \leftarrow$ BuildTree$(V_{a,sep}, E_a)$
$T_b \leftarrow$ BuildTree$(V_{b,sep}, E_b)$
**return** SeparatorTree$(T_a, V_{sep}, T_b)$
```

- Assume we have a fast algorithm to find separators
  - Only puts a vertex in the separator set if it has one edge to each side
  - Separator sets have at least one vertex unless the graph is a clique, in which case the separator contains all but one of the vertices, and that vertex is on the left side of the partition (V_a)
  - For planar graphs this can be done in linear time
- All vertices in the separator set are passed to all children
- Each leaf corresponds to a single edge
- Each leaf includes the indices of its two endpoints and its weight.

SpMat-Vec Multiply: Separator Tree Example



SpMat-Vec Multiply: Multiplication



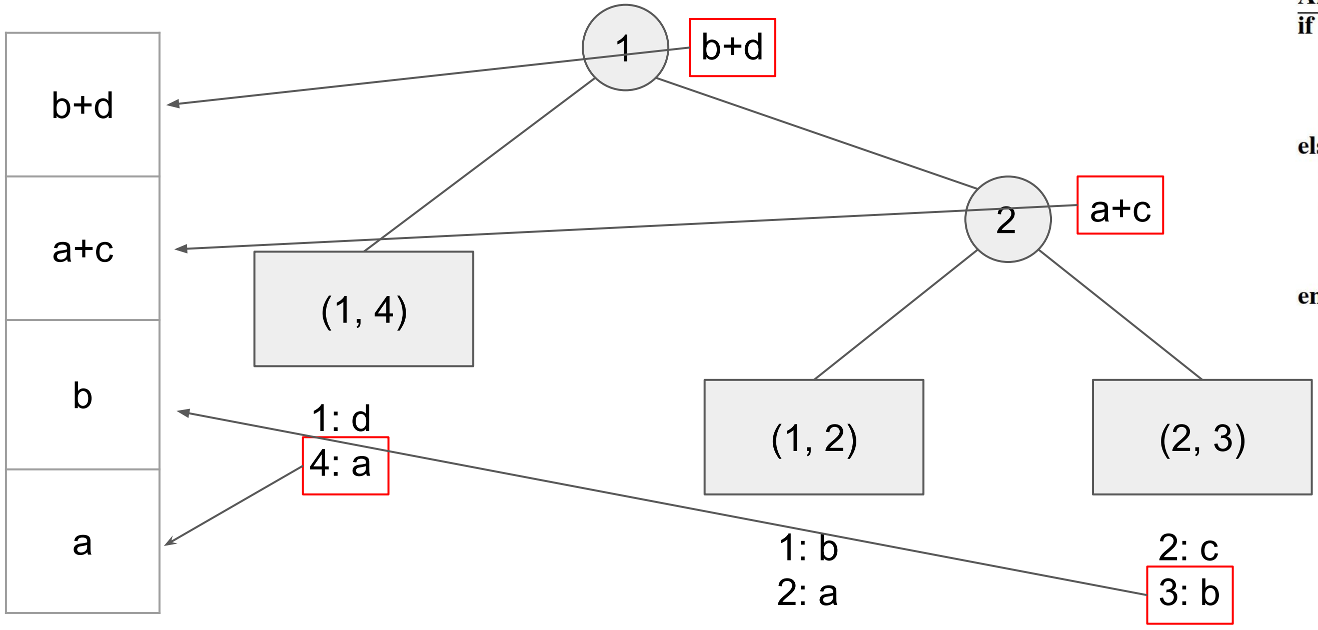
```
**Algorithm** SparseMxV($x$,$T$)
**if** isLeaf($T$) **then**
    $T.u$.value $\leftarrow x[T.v.\text{index}] \otimes T.w_{vu}$
    $T.v$.value $\leftarrow x[T.u.\text{index}] \otimes T.w_{uv}$
    {Two statements for the two edge directions}
**else**
    SparseMxV($T$.left) and SparseMxV($T$.right)
    **for all** $v \in T$.vertices **do**
        $v$.value $\leftarrow$ ($v$.left$\rightarrow$value $\oplus$ $v$.right$\rightarrow$value)
    **end for**
**end if**
```

- Reorder the matrix based on a preorder traversal: all vertices in the top separator will appear first
- Leave the results of multiplications in the root of every vertex
- Whenever it gets to an internal node of a vertex tree it adds the two children.

SpMat-Vec Multiply: Multiplication Example

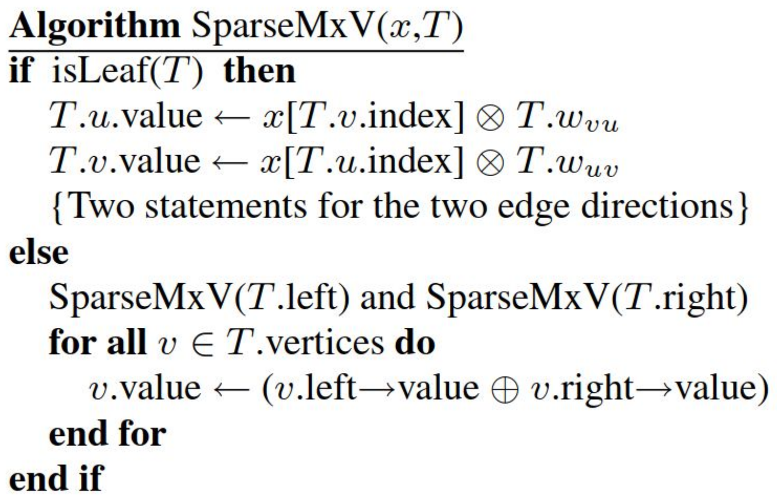
$$\begin{array}{ccccc} & 1 & 2 & 3 & 4 \\ 1 & 0 & 1 & 0 & 1 \\ 2 & 1 & 0 & 1 & 0 \\ 3 & 0 & 1 & 0 & 0 \\ 4 & 1 & 0 & 0 & 0 \end{array} \begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix} = \begin{bmatrix} b+d \\ a+c \\ b \\ a \end{bmatrix}$$

SpMat-Vec Multiply: Multiplication Example

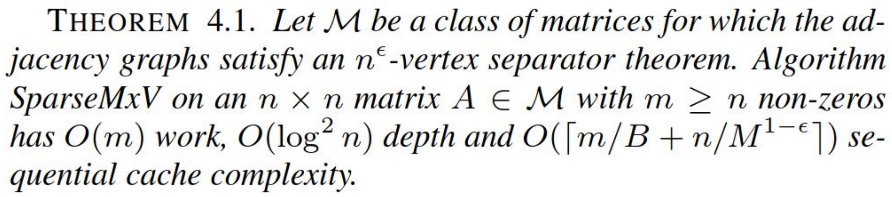


```
Algorithm SparseMxV($x$,$T$)
if isLeaf($T$) then
    $T.u$.value $\leftarrow x[T.v.\text{index}] \otimes T.w_{vu}$
    $T.v$.value $\leftarrow x[T.u.\text{index}] \otimes T.w_{uv}$
    {Two statements for the two edge directions}
else
    SparseMxV($T$.left) and SparseMxV($T$.right)
    for all $v \in T$.vertices do
        $v$.value $\leftarrow$ ($v$.left$\rightarrow$value $\oplus$ $v$.right$\rightarrow$value)
    end for
end if
```

SpMat-Vec Multiply: Multiplication



```
Algorithm SparseMxV($x$,$T$)
if isLeaf($T$) then
    $T.u$.value $\leftarrow x[T.v.\text{index}] \otimes T.w_{vu}$
    $T.v$.value $\leftarrow x[T.u.\text{index}] \otimes T.w_{uv}$
    {Two statements for the two edge directions}
else
    SparseMxV($T$.left) and SparseMxV($T$.right)
    for all $v \in T$.vertices do
        $v$.value $\leftarrow$ ($v$.left$\rightarrow$value $\oplus$ $v$.right$\rightarrow$value)
    end for
end if
```



THEOREM 4.1. *Let $\mathcal{M}$ be a class of matrices for which the adjacency graphs satisfy an n^{ϵ}-vertex separator theorem. Algorithm SparseMxV on an $n \times n$ matrix $A \in \mathcal{M}$ with $m \geq n$ non-zeros has $O(m)$ work, $O(\log^2 n)$ depth and $O(\lceil m/B + n/M^{1-\epsilon} \rceil)$ sequential cache complexity.*

Proof sketch:

- Depth
  - Parallel recursive calls & parallel for
  - Tree: depth O(log(n))
  - For all: depth O(log(n))
  - Total: O(log^2(n))
- Cache complexities:
  - Separate into two cases: heavy vertex copies and light vertex copies
  - Bound the number of heavy vertex copies
  - Bound the number of light vertex copies

Mapping To Parallel Multi-level Hierarchies

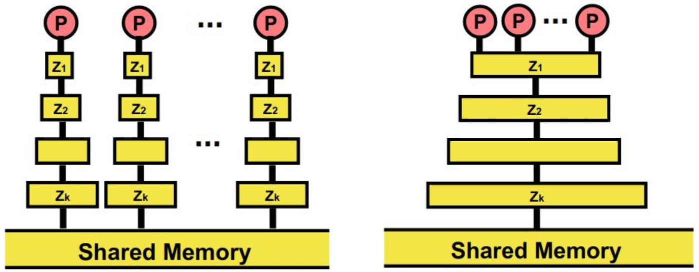


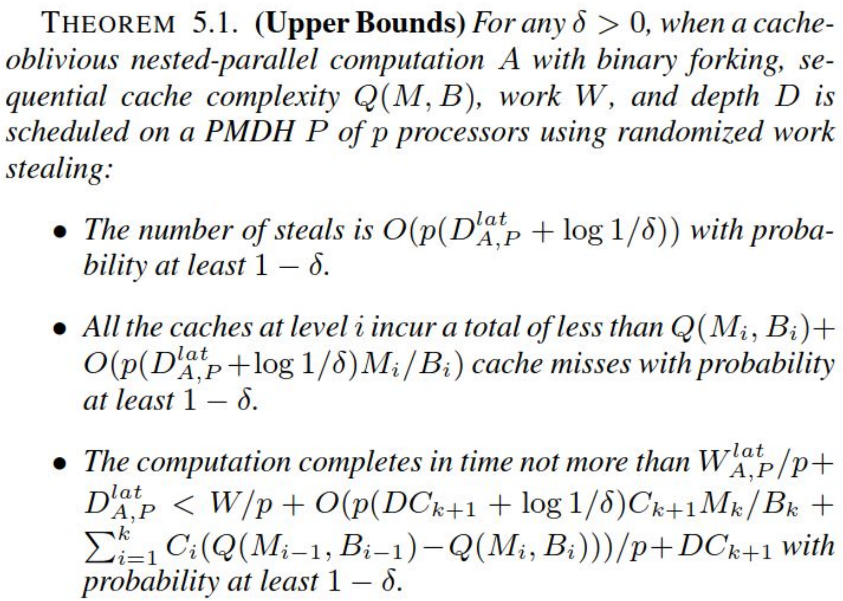
Figure 5: *Left:* Parallel Multi-level Distributed Hierarchy (PMDH). *Right:* Parallel Multi-level Shared Hierarchy (PMSH).

- Cache consistencies in PDMH
  - Caches are non-interfering in that the cache misses of one processor can be analyzed independent of other processors
  - Concurrent reads permitted
  - Concurrent writes (BACKER protocol):
    - If an instruction j is a descendant of instruction i, then values written to memory words by i are reflected in j's memory accesses.
    - Concurrent writes to objects by instructions that do not have a path between them in the dag will not be communicated between processors
    - Reconciled to shared memory and reflected in other cache copies only when a descendant of the instruction that performed these writes tries to access them
  - Reconciliation:
    - Updating all written words within the block. If multiple writes occur, an arbitrary write wins.

PDMH: Work Stealing Scheduler

- Maintains a task dequeue for each processor.
- When a processor spawns a new job, the new job is queued at the tail of its dequeue.
- When a processor runs out of work, it pulls out the job at the head of its task queue. If its own task queue is empty, the processors randomly picks another task queue to steal from.

PDMH: Work Stealing Scheduler



THEOREM 5.1. **(Upper Bounds)** *For any $\delta > 0$, when a cache-oblivious nested-parallel computation A with binary forking, sequential cache complexity $Q(M, B)$, work W, and depth D is scheduled on a PMDH P of p processors using randomized work stealing:*

- *The number of steals is $O(p(D^{lat}_{A,P} + \log 1/\delta))$ with probability at least $1 - \delta$.*
- *All the caches at level i incur a total of less than $Q(M_i, B_i) + O(p(D^{lat}_{A,P} + \log 1/\delta) M_i/B_i)$ cache misses with probability at least $1 - \delta$.*
- *The computation completes in time not more than $W^{lat}_{A,P}/p + D^{lat}_{A,P} < W/p + O(p(DC_{k+1} + \log 1/\delta) C_{k+1} M_k/B_k + \sum_{i=1}^{k} C_i(Q(M_{i-1}, B_{i-1}) - Q(M_i, B_i)))/p + DC_{k+1}$ with probability at least $1 - \delta$.*

Proof sketch:

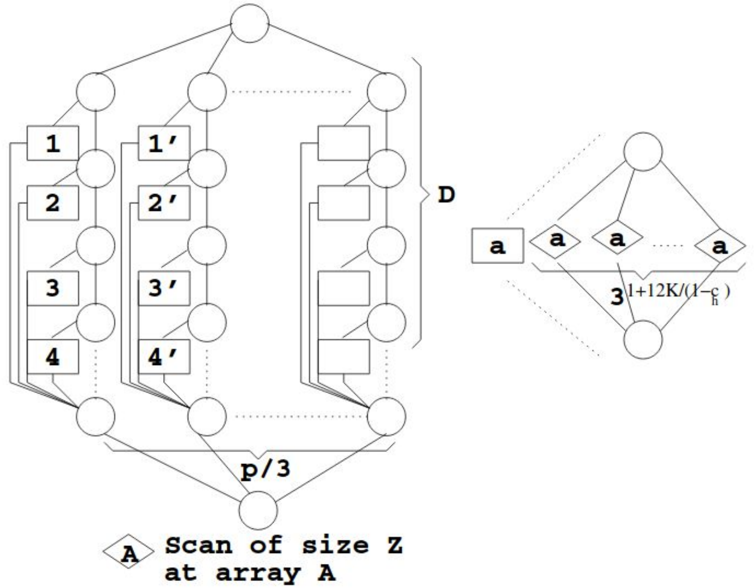
- Reduce the dag to a simpler form
  - Replace instructions with sequences of sequential instructions
  - Each of the replaced instruction take unit time
- Applying Lemma 12 [21] that bounds the number of work-steal attempts
- Use Theorem 13 [21] to bound the running time

[21]: Robert D. Blumofe and Charles E. Leiserson. 1999. Scheduling multithreaded computations by work stealing. J. ACM 46, 5 (Sept. 1999), 720–748.

PDMH: Work Stealing Scheduler

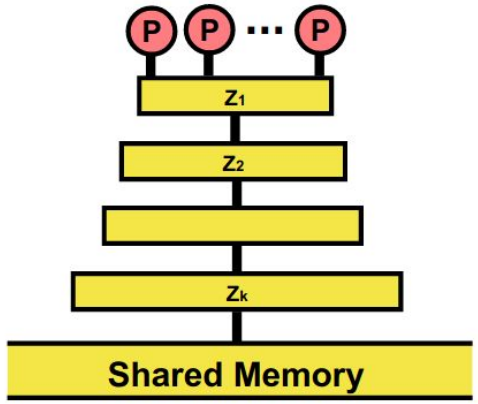
THEOREM 5.2. **(Lower Bound)** *For a PMDH P with any given number of processors $p = \Omega(\log D)$, cache sizes $M_1 < \cdots < M_k \leq M/3$ for some a priori upper bound M, cache line sizes $B_1 \leq \cdots \leq B_k$, and cache latencies $C_1 < \cdots < C_{k+1}$, and for any given depth $D' \geq 3(\log p + \log M) + C_{k+1} + c_0$ (for some constant c_0), we can construct a nested-parallel computation DAG with binary forking and depth D', whose (expected) parallel cache complexity on P, for all levels i, exceeds the sequential cache complexity $Q(M_i, B_i)$ by $\Omega(pD^{lat}_{A,P} M_i/B_i)$ when scheduled using randomized or centralized work stealing.*

Proof sketch:



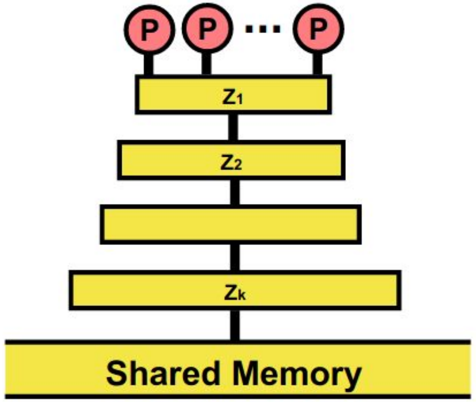
(a) Randomized work stealing

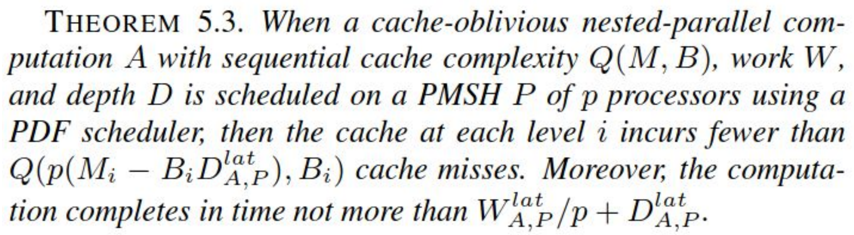
PMSH: Parallel Depth-first Scheduler



- In the PDF scheduler tasks are prioritized according to their ordering in the natural sequential execution
- A processor completing a task is assigned the lowest ranked task among all the available tasks that are ready to execute.
- The relative ranking of available tasks can be efficiently determined on-the-fly without having to perform a sequential execution.
- See:
    - Guy E. Blelloch, Phillip B. Gibbons, and Yossi Matias. 1999. Provably efficient scheduling for languages with fine-grained parallelism. J. ACM 46, 2 (March 1999), 281–321. DOI:https://doi.org/10.1145/301970.301974

PMSH: Parallel Depth-first Scheduler




THEOREM 5.3. *When a cache-oblivious nested-parallel computation A with sequential cache complexity $Q(M, B)$, work W, and depth D is scheduled on a PMSH P of p processors using a PDF scheduler, then the cache at each level i incurs fewer than $Q(p(M_i - B_i D^{lat}_{A,P}), B_i)$ cache misses. Moreover, the computation completes in time not more than $W^{lat}_{A,P}/p + D^{lat}_{A,P}$.*

Proof sketch:

- Generalize results for a single level of shared cache
- Inclusion implies that hits/misses/evictions at levels <i do not alter the number of misses at level i
- Caches sized for inclusion imply that all words in a line evicted at level >i will have already been evicted at level i,

Drawbacks

- Cache complexities depend on low depths
- Assumes cache/DAG consistencies with the BACKER protocol, which is not implemented on real machines
- Assume an optimal cache replacement strategy
    - Can use LRU in practice: cache misses at each level is within a factor of two of the number of misses for a cache half the size running the optimal replacement policy.

Discussion Questions

- Can the proposed sorting algorithm be applied to a distributed environment?
- Is this sparse matrix vector multiplication algorithm fast in practice?
- Will different sparse matrix representations improve/reduce cache complexities of the algorithm proposed?