Funnelselect: Cache-Oblivious Multiple Selection

Sebastian Wild

joint work with Gerth Stølting Brodal



European Symposium on Algorithms 2023

CWI Amsterdam

Outline



1 Multiple Selection



2 Cache Oblivious Algorithms



3 Funnelselect



4 Conclusion


1 Multiple Selection

“What’s this about” in 2min

Two ancient comparison-based **problems** on unordered list of N elements

“What’s this about” in 2min

Two ancient comparison-based **problems** on unordered list of N elements

Sorting

“What’s this about” in 2min

Two ancient comparison-based problems on unordered list of N elements

(Single)
Selection by Rank

e.g. find median

Sorting

“What’s this about” in 2min

Two ancient comparison-based problems on unordered list of N elements

(Single) e.g. find median
Selection by Rank

$O(N)$ comparisons, time
$O(N/B)$ I/Os

internal: Median-of-medians algorithm
IO: Median-of-medians algorithm
CO: Median-of-medians algorithm

Sorting

$O(N \log_2 N)$ comparisons, time
$O(\frac{N}{B} \log_{M/B} \frac{N}{B})$ I/Os

internal: Mergesort
IO: Multiway Mergesort
CO: Funnelsort

“What’s this about” in 2min

Two ancient comparison-based problems on unordered list of N elements

(Single) **Selection** by Rank — e.g. find median

Sorting

more sorted
more expensive

$O(N)$ comparisons, time
$O(N/B)$ I/Os

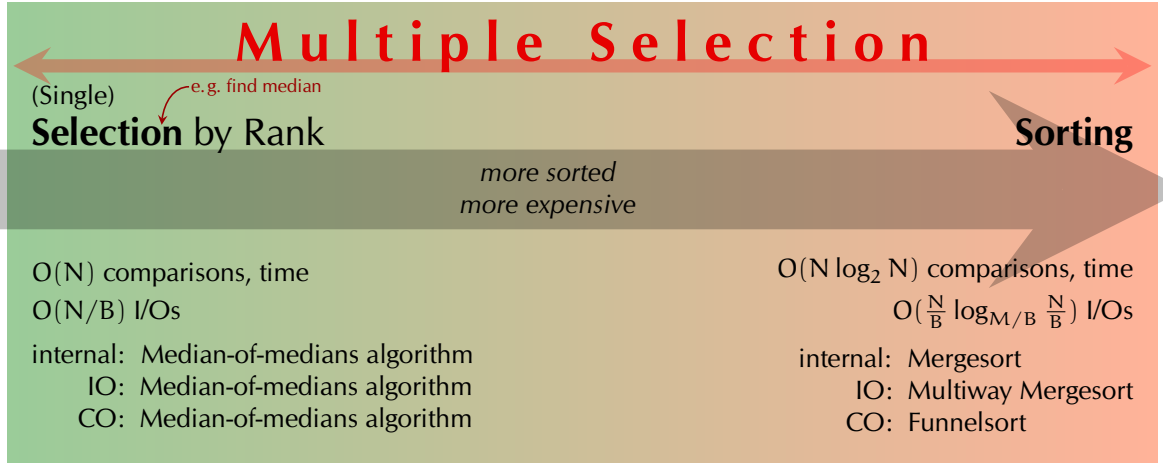
internal: Median-of-medians algorithm
IO: Median-of-medians algorithm
CO: Median-of-medians algorithm

$O(N \log_2 N)$ comparisons, time
$O(\frac{N}{B} \log_{M/B} \frac{N}{B})$ I/Os

internal: Mergesort
IO: Multiway Mergesort
CO: Funnelsort

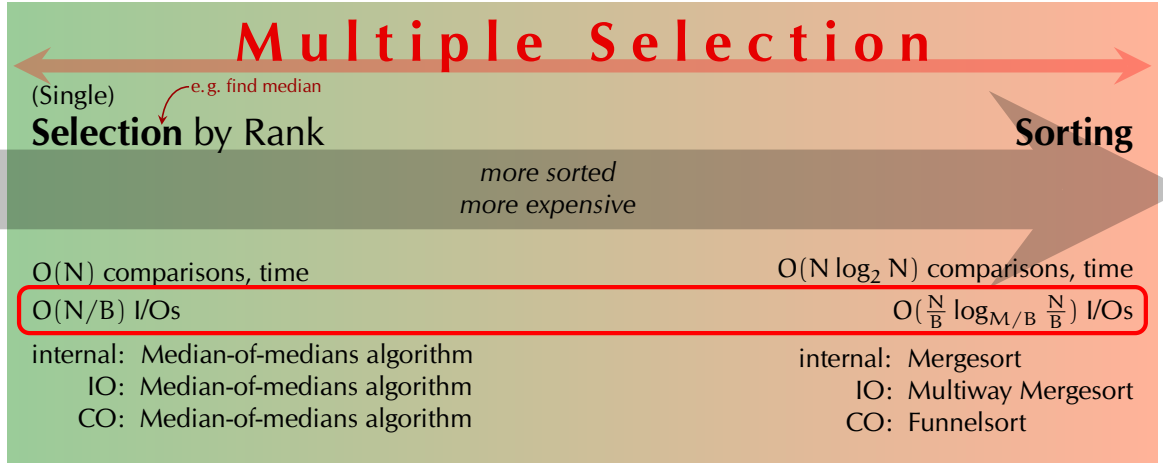
“What’s this about” in 2min

Two ancient comparison-based problems on unordered list of N elements



“What’s this about” in 2min

Two ancient comparison-based problems on unordered list of N elements



This talk: ***What happens between selection and sorting in the cache-oblivious model?***

The Multiple Selection Problem

- **Goal:** find q elements of ranks $r_1 < r_2 < \cdots < r_q$ from unsorted elements $x_1, \ldots, x_N$
  - ⇝ Report sought elements $x_{(r_1)}, \ldots, x_{(r_q)}$ in sorted order

here: all distinct

- **Example:**

x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{10}	x_{11}	x_{12}	x_{13}	x_{14}	x_{15}
67	30	45	33	15	99	26	90	55	9	96	45	95	31	3

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

$r_1 = 1$, $r_2 = 2$, $r_3 = 3$, $r_4 = 8$

- **Simple algorithms:**
  1. q calls to selection algorithm (quickselect, median of medians) ⇝ $O(Nq)$
  2. Divide & conquer 1:
     (single) **select** $r_{\lfloor q/2 \rfloor}$**-th** smallest and partition $x_1, \ldots, x_N$ around it.
     Recursively select $r_1, \ldots, r_{\lfloor q/2 \rfloor - 1}$ and $r_{\lfloor q/2 \rfloor + 1}, \ldots, r_q$ ⇝ $O(N \lg q)$
  3. Divide & conquer 2:
     Find **median of** $x_1, \ldots, x_N$ and split query ranks.
     Recurse where subproblem contains query ranks. ⇝ $O(N \lg q)$

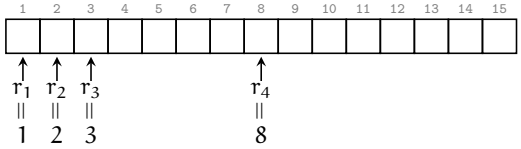
The Multiple Selection Problem

- **Goal:** find q elements of ranks $r_1 < r_2 < \cdots < r_q$ from unsorted elements $x_1, \ldots, x_N$
  - ⇝ Report sought elements $x_{(r_1)}, \ldots, x_{(r_q)}$ in sorted order

here: all distinct

- **Example:**

x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{10}	x_{11}	x_{12}	x_{13}	x_{14}	x_{15}
67	30	45	33	15	99	26	90	55	9	96	45	95	31	3



- **Simple algorithms:**
  1. q calls to selection algorithm (quickselect, median of medians) ⇝ $O(Nq)$
  2. Divide & conquer 1:
     (single) **select** $r_{\lfloor q/2 \rfloor}$**-th** smallest and partition $x_1, \ldots, x_N$ around it.
     Recursively select $r_1, \ldots, r_{\lfloor q/2 \rfloor - 1}$ and $r_{\lfloor q/2 \rfloor + 1}, \ldots, r_q$ ⇝ $O(N \lg q)$
  3. Divide & conquer 2:
     Find **median of** $x_1, \ldots, x_N$ and split query ranks.
     Recurse where subproblem contains query ranks. ⇝ $O(N \lg q)$

The Multiple Selection Problem

- **Goal:** find q elements of ranks $r_1 < r_2 < \cdots < r_q$ from unsorted elements $x_1, \ldots, x_N$
  - ⇝ Report sought elements $x_{(r_1)}, \ldots, x_{(r_q)}$ in sorted order

here: all distinct

- **Example:**

x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{10}	x_{11}	x_{12}	x_{13}	x_{14}	x_{15}
67	30	45	33	15	99	26	90	55	9	96	45	95	31	3

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
3	9	15	26	30	31	33	45	45	55	67	90	95	96	99

$r_1 = 1$, $r_2 = 2$, $r_3 = 3$, $r_4 = 8$

- **Simple algorithms:**
  1. q calls to selection algorithm (quickselect, median of medians) ⇝ $O(Nq)$
  2. Divide & conquer 1:
     (single) **select** $r_{\lfloor q/2 \rfloor}$**-th** smallest and partition $x_1, \ldots, x_N$ around it.
     Recursively select $r_1, \ldots, r_{\lfloor q/2 \rfloor - 1}$ and $r_{\lfloor q/2 \rfloor + 1}, \ldots, r_q$ ⇝ $O(N \lg q)$
  3. Divide & conquer 2:
     Find **median of** $x_1, \ldots, x_N$ and split query ranks.
     Recurse where subproblem contains query ranks. ⇝ $O(N \lg q)$

The Multiple Selection Problem

- **Goal:** find q elements of ranks $r_1 < r_2 < \cdots < r_q$ from unsorted elements $x_1, \ldots, x_N$
  - ⇝ Report sought elements $x_{(r_1)}, \ldots, x_{(r_q)}$ in sorted order

here: all distinct

- **Example:**

x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{10}	x_{11}	x_{12}	x_{13}	x_{14}	x_{15}
67	30	45	33	15	99	26	90	55	9	96	45	95	31	3

Answer: 3, 9, 15, 45

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
3	9	15	26	30	31	33	45	45	55	67	90	95	96	99

$r_1 = 1$, $r_2 = 2$, $r_3 = 3$, $r_4 = 8$

- **Simple algorithms:**
  1. q calls to selection algorithm (quickselect, median of medians) ⇝ $O(Nq)$
  2. Divide & conquer 1:
     (single) **select** $r_{\lfloor q/2 \rfloor}$**-th** smallest and partition $x_1, \ldots, x_N$ around it.
     Recursively select $r_1, \ldots, r_{\lfloor q/2 \rfloor - 1}$ and $r_{\lfloor q/2 \rfloor + 1}, \ldots, r_q$ ⇝ $O(N \lg q)$
  3. Divide & conquer 2:
     Find **median of** $x_1, \ldots, x_N$ and split query ranks.
     Recurse where subproblem contains query ranks. ⇝ $O(N \lg q)$

The Multiple Selection Problem

- **Goal:** find q elements of ranks $r_1 < r_2 < \cdots < r_q$ from unsorted elements $x_1, \ldots, x_N$
  - $\rightsquigarrow$ Report sought elements $x_{(r_1)}, \ldots, x_{(r_q)}$ in sorted order

here: all distinct

- **Example:**

x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{10}	x_{11}	x_{12}	x_{13}	x_{14}	x_{15}
67	30	45	33	15	99	26	90	55	9	96	45	95	31	3

Answer: 3, 9, 15, 45

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
3	9	15	26	30	31	33	45	45	55	67	90	95	96	99

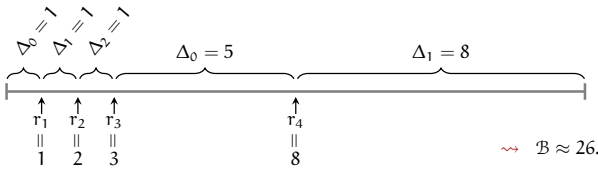
$r_1 = 1$, $r_2 = 2$, $r_3 = 3$, $r_4 = 8$

- **Simple algorithms:**
  1. q **calls** to selection algorithm (quickselect, median of medians) $\rightsquigarrow$ $O(Nq)$
  2. Divide & conquer 1:
     (single) **select $r_{\lceil q/2 \rceil}$-th** smallest and partition $x_1, \ldots, x_N$ around it.
     Recursively select $r_1, \ldots, r_{\lceil q/2 \rceil - 1}$ and $r_{\lceil q/2 \rceil + 1}, \ldots, r_q$ $\rightsquigarrow$ $O(N \lg q)$
  3. Divide & conquer 2:
     Find **median of $x_1, \ldots, x_N$** and split query ranks.
     Recurse where subproblem contains query ranks. $\rightsquigarrow$ $O(N \lg q)$

Fine-grained bounds

“Gap Entropy”:

$$\mathcal{B} = \sum_{i=1}^{q+1} \Delta_i \log_2 \frac{N}{\Delta_i} \qquad \text{with } \Delta_i = r_i - r_{i-1} \qquad (1 \leqslant i \leqslant q+1,\ r_0 = 0 \text{ and } r_{q+1} = N+1)$$



$\rightsquigarrow$ $\mathcal{B} \approx 26.9$

- **lower bound** of $\mathcal{B} - O(N)$ comparisons via “finish sorting” argument
- **upper bound** of $\mathcal{B} + o(\mathcal{B}) + O(N)$ comparisons

Kaligosi, Mehlhorn, Munro & Sanders: *Towards Optimal Multiple Selection*, ICALP **2005**

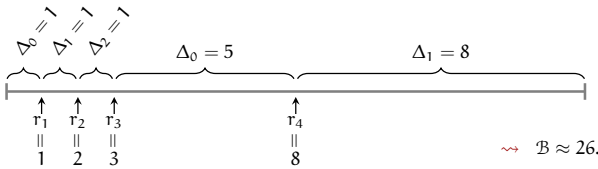
follows from nontrivial analysis;
much easier: repeated median selection & recursion gets $O(\mathcal{B} + N)$ ($\rightarrow$ paper)

$\rightsquigarrow$ In internal memory, multiple-selection problem solved ✓

Fine-grained bounds

“Gap Entropy”:

$$\mathcal{B} = \sum_{i=1}^{q+1} \Delta_i \log_2 \frac{N}{\Delta_i} \qquad \text{with } \Delta_i = r_i - r_{i-1} \qquad (1 \leqslant i \leqslant q+1,\ r_0 = 0 \text{ and } r_{q+1} = N+1)$$



- **lower bound** of $\mathcal{B} - O(N)$ comparisons via “finish sorting” argument

  $\rightsquigarrow$ $\mathcal{B} \approx 26.9$

- **upper bound** of $\mathcal{B} + o(\mathcal{B}) + O(N)$ comparisons

  Kaligosi, Mehlhorn, Munro & Sanders: *Towards Optimal Multiple Selection*, ICALP **2005**

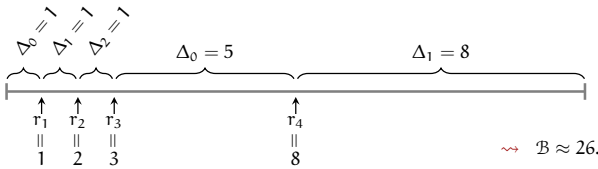
  follows from nontrivial analysis;
  much easier: repeated median selection & recursion gets $O(\mathcal{B} + N)$ ($\rightarrow$ paper)

$\rightsquigarrow$ In internal memory, multiple-selection problem solved ✓

Fine-grained bounds

“Gap Entropy”:

$$\mathcal{B} = \sum_{i=1}^{q+1} \Delta_i \log_2 \frac{N}{\Delta_i} \qquad \text{with } \Delta_i = r_i - r_{i-1} \qquad (1 \leqslant i \leqslant q+1,\ r_0 = 0 \text{ and } r_{q+1} = N+1)$$



- **lower bound** of $\mathcal{B} - O(N)$ comparisons via “finish sorting” argument

  ⇝ $\mathcal{B} \approx 26.9$

- **upper bound** of $\mathcal{B} + o(\mathcal{B}) + O(N)$ comparisons

  follows from nontrivial analysis;
  much easier: repeated median selection & recursion gets $O(\mathcal{B} + N)$ (→ paper)

Kaligosi, Mehlhorn, Munro & Sanders: *Towards Optimal Multiple Selection*, ICALP **2005**

⇝ In internal memory, multiple-selection problem solved ✓

Outline



1 Multiple Selection



2 Cache Oblivious Algorithms



3 Funnelselect

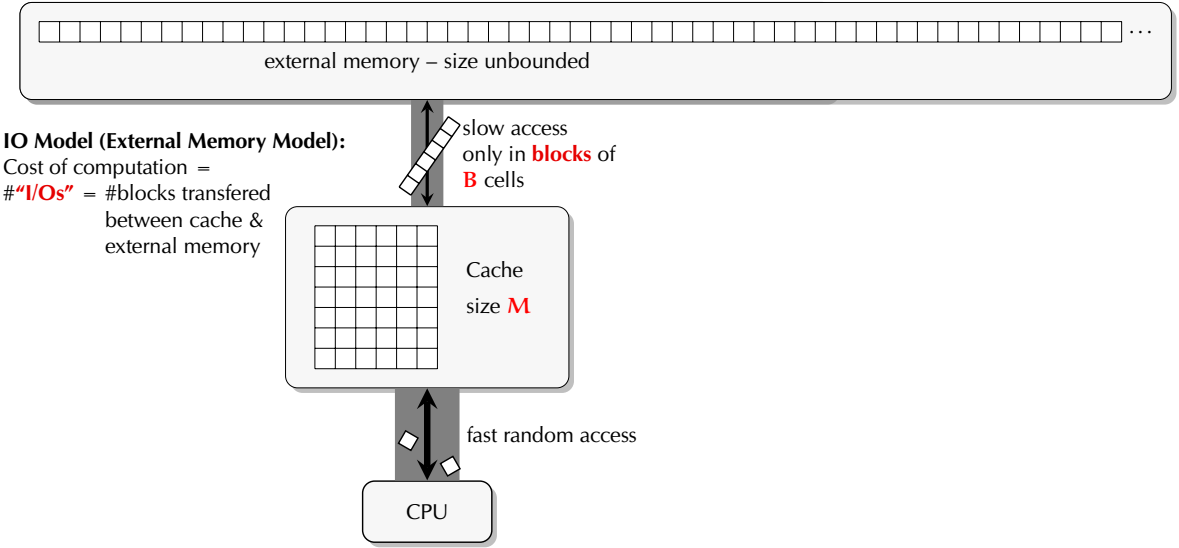


4 Conclusion



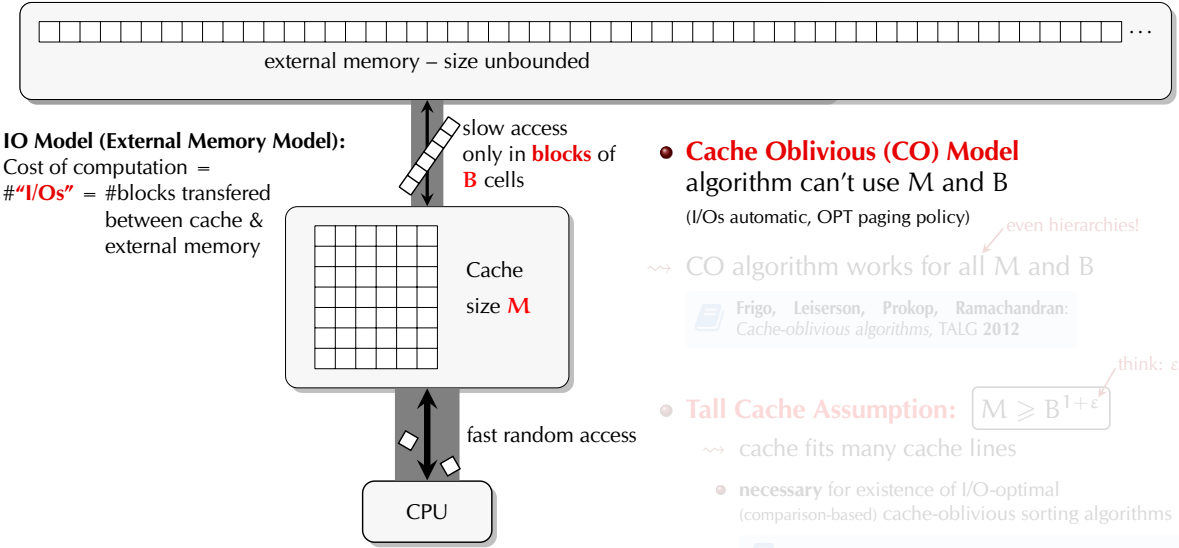
2 Cache Oblivious Algorithms

IO Model & CO Model



IO Model (External Memory Model):
Cost of computation =
#**"I/Os"** = #blocks transfered between cache & external memory

IO Model & CO Model

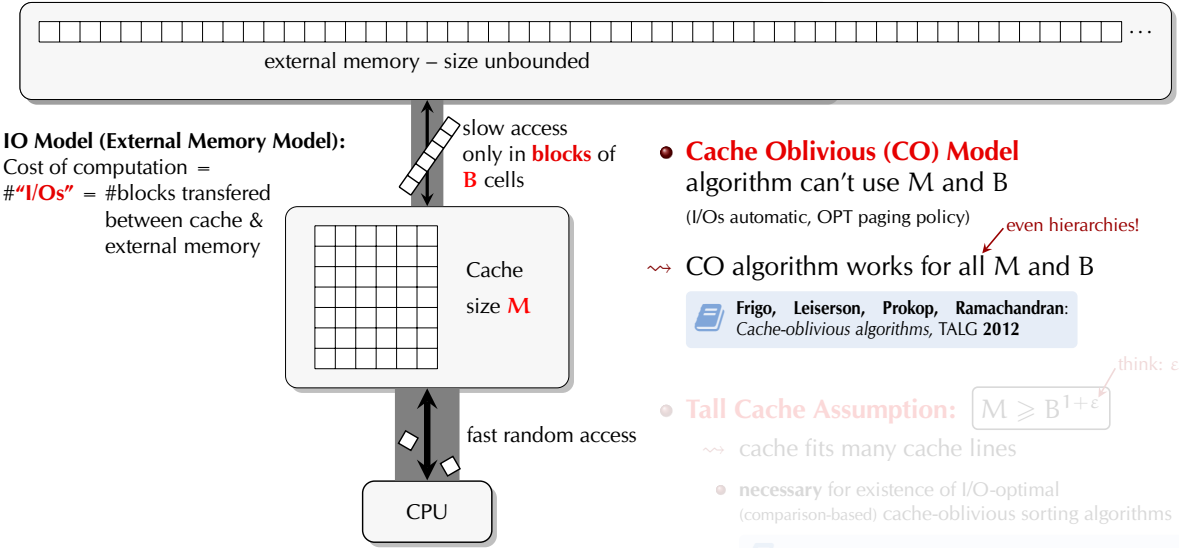


IO Model (External Memory Model):
Cost of computation = #"I/Os" = #blocks transfered between cache & external memory

- **Cache Oblivious (CO) Model**
  algorithm can't use M and B
  (I/Os automatic, OPT paging policy)
  even hierarchies!
  ⇝ CO algorithm works for all M and B
  Frigo, Leiserson, Prokop, Ramachandran: *Cache-oblivious algorithms*, TALG **2012**
- **Tall Cache Assumption:** $M \geqslant B^{1+\varepsilon}$ (think: $\varepsilon = 1$)
  ⇝ cache fits many cache lines
  - **necessary** for existence of I/O-optimal (comparison-based) cache-oblivious sorting algorithms

  Brodal, Fagerberg: *On the limits of cache-obliviousness*, STOC **2003**

IO Model & CO Model



IO Model (External Memory Model):
Cost of computation = #"I/Os" = #blocks transfered between cache & external memory

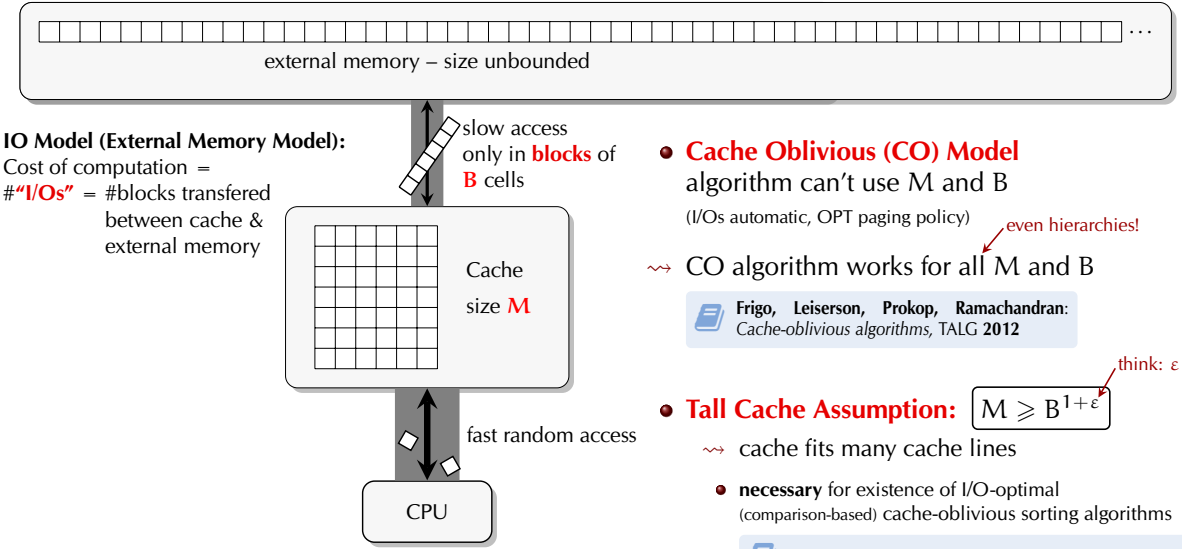
- **Cache Oblivious (CO) Model**
  algorithm can't use M and B
  (I/Os automatic, OPT paging policy)
  even hierarchies!
  ⇝ CO algorithm works for all M and B

  Frigo, Leiserson, Prokop, Ramachandran: *Cache-oblivious algorithms*, TALG **2012**

- **Tall Cache Assumption:** $M \geqslant B^{1+\varepsilon}$ (think: $\varepsilon = 1$)
  ⇝ cache fits many cache lines
  - **necessary** for existence of I/O-optimal (comparison-based) cache-oblivious sorting algorithms

  Brodal, Fagerberg: *On the limits of cache-obliviousness*, STOC **2003**

IO Model & CO Model



IO Model (External Memory Model):
Cost of computation =
#"I/Os" = #blocks transfered between cache & external memory

- **Cache Oblivious (CO) Model**
  algorithm can't use M and B
  (I/Os automatic, OPT paging policy)
  even hierarchies!
  ⇝ CO algorithm works for all M and B

  Frigo, Leiserson, Prokop, Ramachandran: *Cache-oblivious algorithms,* TALG **2012**

- **Tall Cache Assumption:** $M \geqslant B^{1+\varepsilon}$ (think: $\varepsilon = 1$)
  ⇝ cache fits many cache lines
  - **necessary** for existence of I/O-optimal (comparison-based) cache-oblivious sorting algorithms

  Brodal, Fagerberg: *On the limits of cache-obliviousness,* STOC **2003**

Previous Work & Our Results

Reference		Comparisons	I/Os	Comments
Single selection				
Blum *et al.* 1973	wc	5.4305N	$O(N/B)$	CO, deterministic

Previous Work & Our Results

Reference		Comparisons	I/Os	Comments
Single selection				
Blum *et al.* 1973	wc	$5.4305N$	$O(N/B)$	CO, deterministic
Multiple selection				
Dobkin & Munro 1981	wc	$3\mathcal{B} + O(N)$	$O((\mathcal{B}+N)/B)$	CO, deterministic
Hu *et al.* 2014	wc	$O(\mathcal{B}+N)$	$O(\mathcal{B}_{I/O}+N/B)$	deterministic
Barbay *et al.* 2016	wc	$O(\mathcal{B}+N)$	$O(\mathcal{B}_{I/O}+N/B)$	online, determ., $M \geqslant B^{1+\varepsilon}$

$$\mathcal{B} = \sum_{i=1}^{q+1} \Delta_i \log_2\left(\frac{N}{\Delta_i}\right) \qquad \mathcal{B}_{I/O} = \sum_{i=1}^{q+1} \frac{\Delta_i}{B} \log_{\frac{M}{B}}\left(\frac{N}{\Delta_i}\right) = \frac{\mathcal{B}}{B\log_2(M/B)} \underset{\text{(usually)}}{\ll} \frac{\mathcal{B}}{B}$$

Previous Work & Our Results

Reference		Comparisons	I/Os	Comments
Single selection				
Blum *et al.* 1973	wc	$5.4305N$	$O(N/B)$	CO, deterministic
Multiple selection				
Dobkin & Munro 1981	wc	$3\mathcal{B} + O(N)$	$O((\mathcal{B}+N)/B)$	CO, deterministic
Hu *et al.* 2014	wc	$O(\mathcal{B}+N)$	$O(\mathcal{B}_{I/O} + N/B)$	deterministic
Barbay *et al.* 2016	wc	$O(\mathcal{B}+N)$	$O(\mathcal{B}_{I/O} + N/B)$	online, determ., $M \geqslant B^{1+\varepsilon}$
Funnelselect NEW	$\mathbb{E}$	$O(\mathcal{B}+N)$	$\mathbf{O(\mathcal{B}_{I/O} + N/B)}$	**CO**, randomized, $M \geqslant B^{1+\varepsilon}$
Lower bound		$\mathcal{B} - O(N)$	$\mathbf{\Omega(\mathcal{B}_{I/O})} - O(\frac{N}{B})$	$M \geqslant B^{1+\varepsilon}$

$$\mathcal{B} = \sum_{i=1}^{q+1} \Delta_i \log_2\left(\frac{N}{\Delta_i}\right)$$

$$\mathcal{B}_{I/O} = \sum_{i=1}^{q+1} \frac{\Delta_i}{B} \log_{\frac{M}{B}}\left(\frac{N}{\Delta_i}\right) = \frac{\mathcal{B}}{B \log_2(M/B)} \underset{\text{(usually)}}{\ll} \frac{\mathcal{B}}{B}$$

Previous Work & Our Results

Reference		Comparisons	I/Os	Comments
Single selection				
Blum *et al.* 1973	wc	$5.4305N$	$O(N/B)$	CO, deterministic
Multiple selection				
Dobkin & Munro 1981	wc	$3\mathcal{B} + O(N)$	$O((\mathcal{B}+N)/B)$	CO, deterministic
Hu *et al.* 2014	wc	$O(\mathcal{B}+N)$	$O(\mathcal{B}_{I/O} + N/B)$	deterministic
Barbay *et al.* 2016	wc	$O(\mathcal{B}+N)$	$O(\mathcal{B}_{I/O} + N/B)$	online, determ., $M \geqslant B^{1+\varepsilon}$
Funnelselect NEW	$\mathbb{E}$	$O(\mathcal{B}+N)$	$\mathbf{O(\mathcal{B}_{I/O} + N/B)}$	**CO**, randomized, $M \geqslant B^{1+\varepsilon}$
Lower bound		$\mathcal{B} - O(N)$	$\mathbf{\Omega(\mathcal{B}_{I/O})} - O\left(\frac{N}{B}\right)$	$M \geqslant B^{1+\varepsilon}$

$$\mathcal{B} = \sum_{i=1}^{q+1} \Delta_i \log_2\left(\frac{N}{\Delta_i}\right) \qquad \mathcal{B}_{I/O} = \sum_{i=1}^{q+1} \frac{\Delta_i}{B} \log_{\frac{M}{B}}\left(\frac{N}{\Delta_i}\right) = \frac{\mathcal{B}}{B \log_2(M/B)} \underset{\text{(usually)}}{\ll} \frac{\mathcal{B}}{B}$$

Funnelselect is the first cache-oblivious I/O-optimal algorithm.

Our technical challenge

1. **Key algorithmic change from internal $\rightsquigarrow$ external-memory:**
   (for sorting and multiple selection)

   binary partitioning $\rightsquigarrow$ $\approx \frac{M}{B}$-way partitioning

   ↯ can't do that cache obliviously

2. **cache-oblivious sorting** uses N^{δ}-way partitioning by "funnels"

   $\rightsquigarrow$ first partitioning round already has sorting complexity $\rightsquigarrow$ $\omega(\mathcal{B}_{I/O})$ I/Os

   ↯ can't use that for multiple selection

Our technical challenge

1. **Key algorithmic change from internal $\rightsquigarrow$ external-memory:**
   (for sorting and multiple selection)

   binary partitioning $\rightsquigarrow$ $\approx \frac{M}{B}$-way partitioning

   ↯ can't do that cache obliviously

2. **cache-oblivious sorting** uses N^{δ}-way partitioning by "funnels"

   $\rightsquigarrow$ first partitioning round already has sorting complexity $\rightsquigarrow$ $\omega(\mathcal{B}_{I/O})$ I/Os

   ↯ can't use that for multiple selection

Our technical challenge

1. **Key algorithmic change from internal $\rightsquigarrow$ external-memory:**
   (for sorting and multiple selection)

   binary partitioning $\rightsquigarrow$ $\approx \frac{M}{B}$-way partitioning

   ↯ can't do that cache obliviously

2. **cache-oblivious sorting** uses N^{δ}-way partitioning by "funnels"

   $\rightsquigarrow$ first partitioning round already has sorting complexity $\rightsquigarrow$ $\omega(\mathcal{B}_{I/O})$ I/Os

   ↯ can't use that for multiple selection

Our technical challenge

1. **Key algorithmic change from internal $\rightsquigarrow$ external-memory:**
   (for sorting and multiple selection)

   $$\boxed{\text{binary partitioning} \rightsquigarrow \approx \tfrac{M}{B}\text{-way partitioning}}$$

   ↯ can't do that cache obliviously

2. **cache-oblivious sorting** uses $\boxed{N^{\delta}\text{-way partitioning by "funnels"}}$

   $\rightsquigarrow$ first partitioning round already has sorting complexity $\rightsquigarrow$ $\omega(\mathcal{B}_{I/O})$ I/Os

   ↯ can't use that for multiple selection

Outline



1 Multiple Selection



2 Cache Oblivious Algorithms



3 Funnelselect



4 Conclusion



3 Funnelselect

Funnels for Partitioning



Can use funnels from Funnelsort in reverse!

Funnels for Partitioning



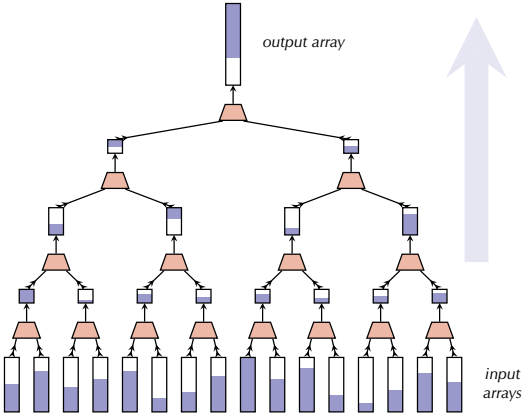
Can use funnels from Funnelsort in reverse!

Funnel:

- $k = \sqrt[4]{N}$-way merger (recursive binary merges)
- **judiciously sized buffers** for intermediate results

Simplifying assumptions:

$N = 2^{2^i}$ and $d = 4$



Funnels for Partitioning



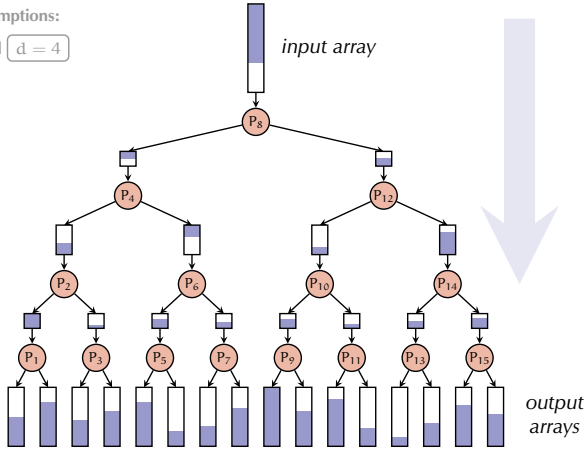
Can use funnels from Funnelsort in reverse!

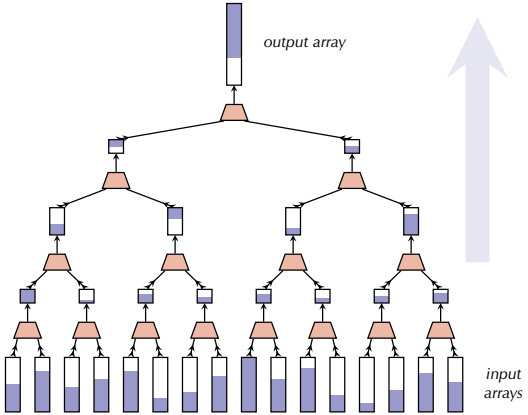
Funnel:

- $k = \sqrt[4]{N}$-way merger (recursive binary merges)
- **judiciously sized buffers** for intermediate results

Simplifying assumptions:

$N = 2^{2^i}$ and $d = 4$





Funnels for Partitioning



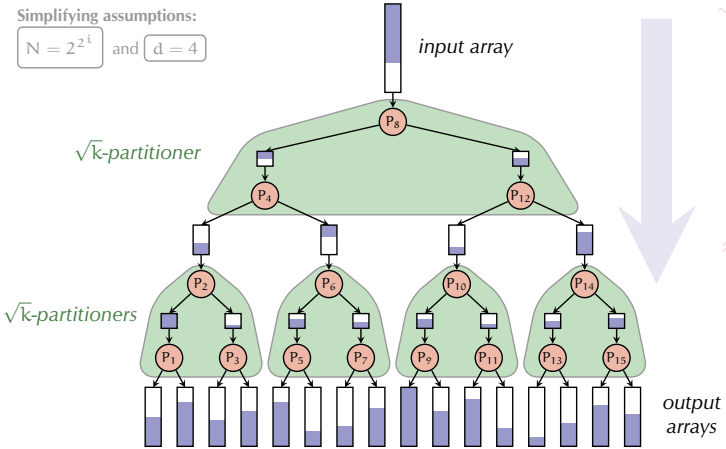
Can use funnels from Funnelsort in reverse!

Funnel:

- $k = \sqrt[4]{N}$-way merger (recursive binary merges)
- **judiciously sized buffers** for intermediate results

Simplifying assumptions:

$N = 2^{2^i}$ and $d = 4$



k-partitioner

- Funnel recursion (van Emde Boas)
- Funnel buffer sizes (largest in middle layer)

⇝ overall space $O(k^{5/2}) = O(N^{5/8})$

- Nodes partition around **pivot value P**
- Partition = **push** input down
  - when output buffer full, **recursively** push
  - need final flush of all buffers

≈ ***I/O-optimal cache-oblivious gadget***
for $\hat{k}$-way **partitioning** with $\hat{k} \approx M^{0.3}$

⇝ Can be used for expected I/O-optimal CO quicksort
(but better options available)

Funnels for Partitioning



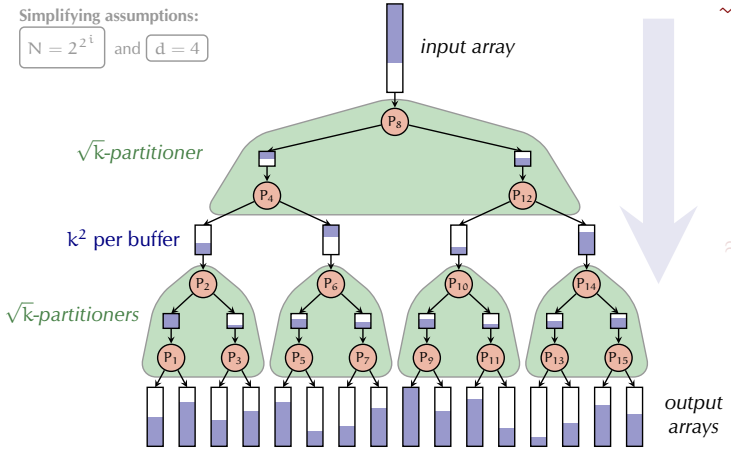
Can use funnels from Funnelsort in reverse!

Funnel:

- $k = \sqrt[4]{N}$-way merger (recursive binary merges)
- **judiciously sized buffers** for intermediate results

Simplifying assumptions:

$N = 2^{2^i}$ and $d = 4$



k-partitioner

- Funnel recursion (van Emde Boas)
- Funnel buffer sizes (largest in middle layer)

⇝ overall space $O(k^{5/2}) = O(N^{5/8})$

- Nodes partition around **pivot value P**
- Partition = **push** input down
  - when output buffer full, **recursively** push
  - need final flush of all buffers

≈ ***I/O-optimal cache-oblivious gadget***
for $\hat{k}$-way **partitioning** with $\hat{k} \approx M^{0.3}$

⇝ Can be used for expected I/O-optimal CO quicksort
(but better options available)

Funnels for Partitioning



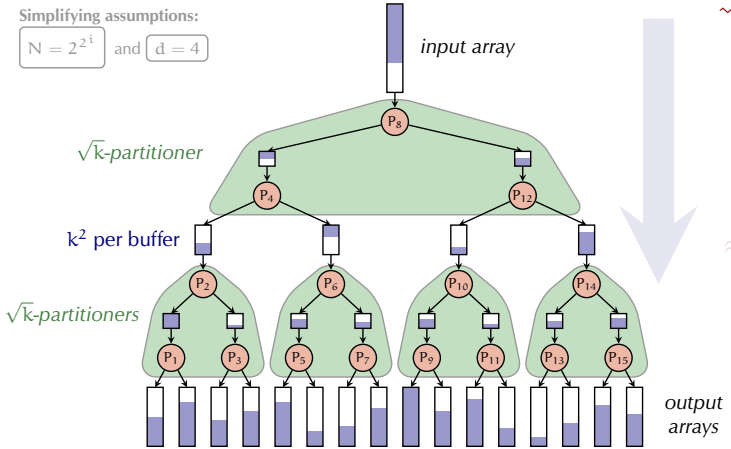
Can use funnels from Funnelsort in reverse!

Funnel:

- $k = \sqrt[4]{N}$-way merger (recursive binary merges)
- **judiciously sized buffers** for intermediate results

Simplifying assumptions:

$N = 2^{2^i}$ and $d = 4$



k-partitioner

- Funnel recursion (van Emde Boas)
- Funnel buffer sizes (largest in middle layer)

$\rightsquigarrow$ overall space $O(k^{5/2}) = O(N^{5/8})$

- Nodes partition around **pivot value P**
- Partition = **push** input down
  - when output buffer full, **recursively** push
  - need final flush of all buffers

$\approx$ ***I/O-optimal cache-oblivious gadget***
for $\hat{k}$-way **partitioning** with $\hat{k} \approx M^{0.3}$

$\rightsquigarrow$ Can be used for expected I/O-optimal CO quicksort
(but better options available)

Funnels for Partitioning



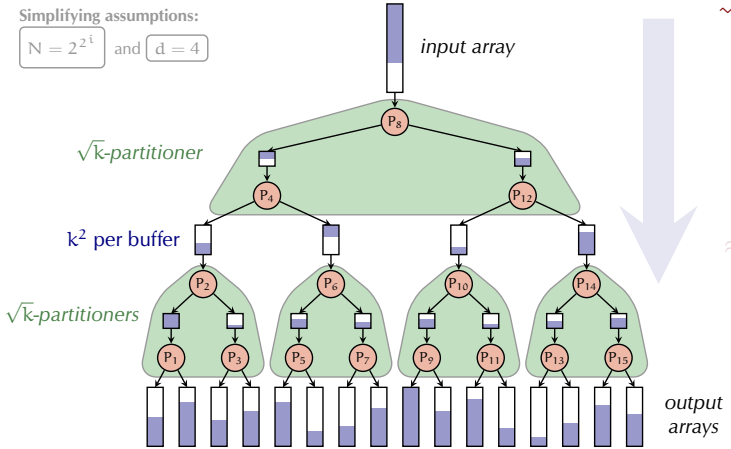
Can use funnels from Funnelsort in reverse!

Funnel:

- $k = \sqrt[4]{N}$-way merger (recursive binary merges)
- **judiciously sized buffers** for intermediate results

Simplifying assumptions:

$N = 2^{2^i}$ and $d = 4$



k-partitioner

- Funnel recursion (van Emde Boas)
- Funnel buffer sizes (largest in middle layer)

$\rightsquigarrow$ overall space $O(k^{5/2}) = O(N^{5/8})$

- Nodes partition around **pivot value P**
- Partition = **push** input down
  - when output buffer full, **recursively** push
  - need final flush of all buffers

$\approx$ ***I/O-optimal cache-oblivious gadget***
for $\hat{k}$-way **partitioning** with $\hat{k} \approx M^{0.3}$

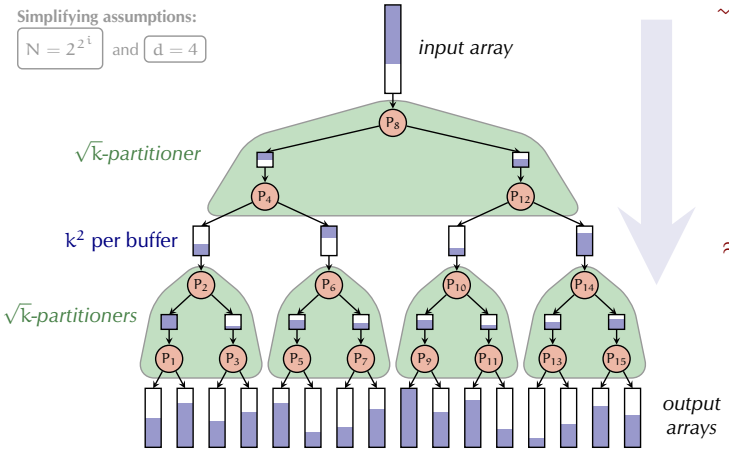
$\rightsquigarrow$ Can be used for expected I/O-optimal CO quicksort
(but better options available)

Funnels for Partitioning

Can use funnels from Funnelsort in reverse!

Funnel:

- $k = \sqrt[4]{N}$-way merger (recursive binary merges)
- **judiciously sized buffers** for intermediate results



k-partitioner

- Funnel recursion (van Emde Boas)
- Funnel buffer sizes (largest in middle layer)

$\rightsquigarrow$ overall space $O(k^{5/2}) = O(N^{5/8})$

- Nodes partition around **pivot value P**
- Partition = **push** input down
  - when output buffer full, **recursively** push
  - need final flush of all buffers

$\approx$ ***I/O-optimal cache-oblivious gadget***
for $\hat{k}$-way **partitioning** with $\hat{k} \approx M^{0.3}$

$\rightsquigarrow$ Can be used for expected I/O-optimal CO quicksort
(but better options available)

Early Truncation

- **Recall:** for multiple selection (in general), can't use full k-partitioner
  (already sorting complexity)

Early truncation: *Don't split buckets that don't contain any query ranks!*

Buckets depend on (random) pivots

⇝ Only know a query's bucket **after** partitioning . . .

within safety margin $\pm\xi = N^{1/2+\delta}$

Assume bucket boundaries **close** to **expected** location.

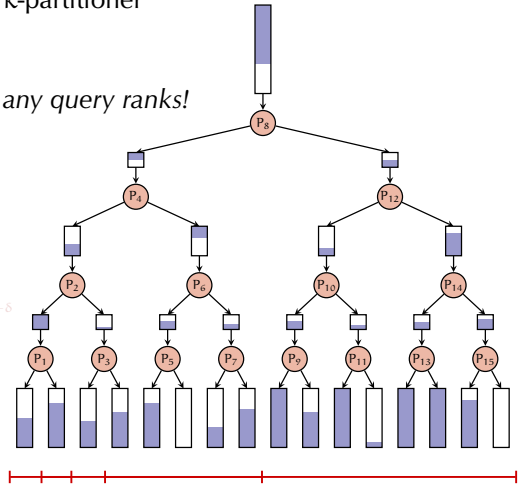
⇝ If bucket is ***expected query free***, don't split further.

- "Expected query free" known up front
  (Depends only on N & query ranks)
  ⇝ Can remove unwanted partitioning nodes from funnel in preprocessing.

Early Truncation

- **Recall:** for multiple selection (in general), can't use full k-partitioner
  (already sorting complexity)

Early truncation: *Don't split buckets that don't contain any query ranks!*



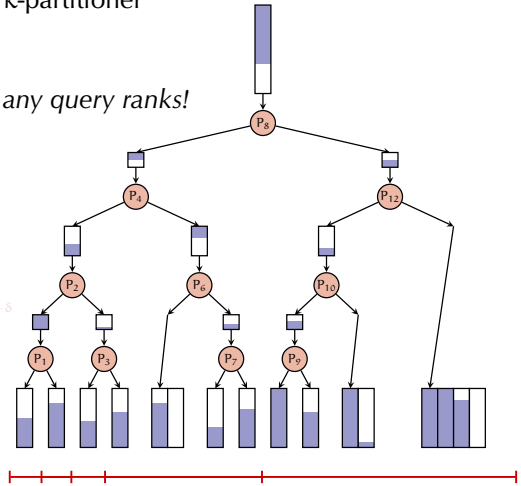
- Buckets depend on (random) pivots
- ⇝ Only know a query's bucket **after** partitioning . . .

- **Assume** bucket boundaries **close** (within safety margin $\pm\xi = N^{1/2+\delta}$) to **expected** location.
- ⇝ If bucket is ***expected query free***, don't split further.
  - "Expected query free" known up front
    (Depends only on N & query ranks)
  - ⇝ Can remove unwanted partitioning nodes from funnel in preprocessing.

Early Truncation

- **Recall:** for multiple selection (in general), can't use full k-partitioner
  (already sorting complexity)

Early truncation: *Don't split buckets that don't contain any query ranks!*



- Buckets depend on (random) pivots
- ⇝ Only know a query's bucket **after** partitioning . . .

within safety margin $\pm\xi = N^{1/2+\delta}$

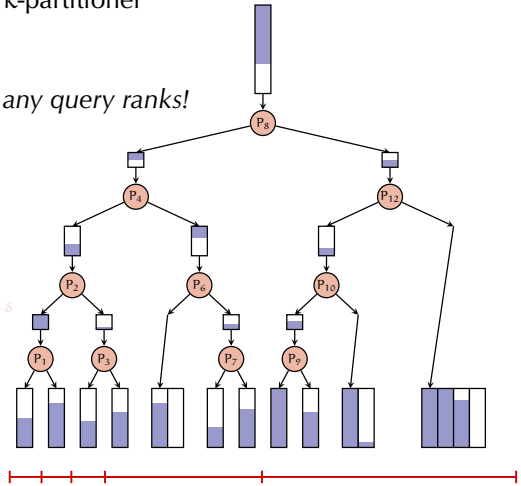
- **Assume** bucket boundaries **close** to **expected** location.
- ⇝ If bucket is ***expected query free***, don't split further.
  - "Expected query free" known up front
    (Depends only on N & query ranks)
  - ⇝ Can remove unwanted partitioning nodes from funnel in preprocessing.

Early Truncation

- **Recall:** for multiple selection (in general), can't use full k-partitioner
  (already sorting complexity)



Early truncation: *Don't split buckets that don't contain any query ranks!*



- Buckets depend on (random) pivots
- ⇝ Only know a query's bucket **after** partitioning …

within safety margin $\pm\xi = N^{1/2+\delta}$

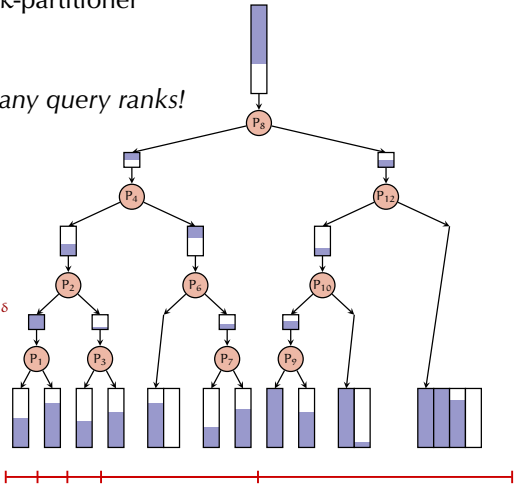
Assume bucket boundaries **close** to **expected** location.

- ⇝ If bucket is ***expected query free***, don't split further.
  - "Expected query free" known up front
    (Depends only on N & query ranks)
  - ⇝ Can remove unwanted partitioning nodes from funnel in preprocessing.

Early Truncation

- **Recall:** for multiple selection (in general), can't use full k-partitioner
  (already sorting complexity)

Early truncation: *Don't split buckets that don't contain any query ranks!*



- Buckets depend on (random) pivots

⇝ Only know a query's bucket **after** partitioning …

within safety margin $\pm\xi = N^{1/2+\delta}$

Assume bucket boundaries **close** to **expected** location.

⇝ If bucket is ***expected query free***, don't split further.

- "Expected query free" known up front
  (Depends only on N & query ranks)

⇝ Can remove unwanted partitioning nodes from funnel in preprocessing.

Pivot Sampling

- Recall this?



Assume bucket boundaries **close** to **expected** location.

within safety margin $\pm\xi_i = N^{1/2+\delta}$

- We have to *make* that true by choosing pivots well

The following randomized choice works *with high probability:* (by standard Chernoff bound arguments)

1. Include each element in sample $\bar{S}$ with prob. $p = 1/\log_2(N)$.
2. Sort the sample $\bar{S}$.
3. Pick pivot P_i as $\approx ipN/k$th smallest in $\bar{S}$ $\quad (i = 1, \ldots, k-1)$

Pivot Sampling

- Recall this?



Assume bucket boundaries **close** to **expected** location.

within safety margin $\pm\xi_i = N^{1/2+\delta}$

- We have to *make* that true by choosing pivots well



The following randomized choice works *with high probability*: (by standard Chernoff bound arguments)

1. Include each element in sample $\bar{S}$ with prob. $p = 1/\log_2(N)$.
2. Sort the sample $\bar{S}$.
3. Pick pivot P_i as $\approx ipN/k$th smallest in $\bar{S}$ $\quad (i = 1, \ldots, k-1)$

Overall Algorithm

Funnelselect:

1. Sample pivots $P_1, \ldots, P_{k-1}$
2. Build k-partitioner using $P_1, \ldots, P_{k-1}$
3. Mark expected query free buckets & rewire their parent's buffer to output
4. For each bucket with queries:
   - i. Sort the bucket (Funnelsort)
   - ii. Report sought elements

Observations:

- No (top-level) recursion needed
- Algorithm can *fail* at several places ⇝ restart
  - but: with high probability, no fails
  - ⇝ no effect on expected running time

sample too small,
sample too large,
pivots too skewed,
query in expected
query free bucket

- Can be augmented to produce input **partitioned** around sought elements in contiguous external memory
  (default: only return sought elements in order)
- Can be made to handle equal elements

Overall Algorithm

Funnelselect:

1. Sample pivots $P_1, \ldots, P_{k-1}$
2. Build k-partitioner using $P_1, \ldots, P_{k-1}$
3. Mark expected query free buckets & rewire their parent's buffer to output
4. For each bucket with queries:
   - i. Sort the bucket (Funnelsort)
   - ii. Report sought elements

Observations:

- No (top-level) recursion needed
- Algorithm can *fail* at several places ⇝ restart
  - but: with high probability, no fails
  - ⇝ no effect on expected running time

sample too small,
sample too large,
pivots too skewed,
query in expected
query free bucket

- Can be augmented to produce input **partitioned** around sought elements in contiguous external memory
  (default: only return sought elements in order)
- Can be made to handle equal elements

Outline



1 Multiple Selection



2 Cache Oblivious Algorithms



3 Funnelselect



4 Conclusion



4 Conclusion

Conclusion

- We presented ✨ ***Funnelselect*** ✨
  the first **I/O-optimal, cache-oblivious multiple-selection algorithm**
- The presented algorithm is inherently **randomized**,
  but we meanwhile found a deterministic method! (but too late for proceedings . . . but stay tuned)

Open Problems:

1. Funnelselect assumes a **tall cache**.
   Cannot be avoided since I/O-opt. CO sorting ($\subset$ multiple selection) requires it.
   But single selection doesn't! $\rightsquigarrow$ What happens in between?
2. Can ***online*** multiple selection be solved I/O-optimal cache obliviously?

Conclusion

- We presented ✨ ***Funnelselect*** ✨
  the first **I/O-optimal, cache-oblivious multiple-selection algorithm**
- The presented algorithm is inherently **randomized**,
  but we meanwhile found a deterministic method! (but too late for proceedings … but stay tuned)

Open Problems:

1. Funnelselect assumes a **tall cache**.
   Cannot be avoided since I/O-opt. CO sorting ($\subset$ multiple selection) requires it.
   But single selection doesn't! $\rightsquigarrow$ What happens in between?
2. Can ***online*** multiple selection be solved I/O-optimal cache obliviously?

Conclusion

- We presented ✨ ***Funnelselect*** ✨
  the first **I/O-optimal, cache-oblivious multiple-selection algorithm**
- The presented algorithm is inherently **randomized**,
  but we meanwhile found a deterministic method! (but too late for proceedings … but stay tuned)

Open Problems:

1. Funnelselect assumes a **tall cache**.
   Cannot be avoided since I/O-opt. CO sorting ($\subset$ multiple selection) requires it.
   But single selection doesn't! $\rightsquigarrow$ What happens in between?
2. Can ***online*** multiple selection be solved I/O-optimal cache obliviously?

We're hiring! for
Computing over compressed graph-structured data

- 3 year **postdoc**
- **PhD student**

Liverpool sounds cool? $\rightarrow$ Talk to me!



Icons made by *Freepik*, *Gregor Cresnar*, *Those Icons*, *Smashicons*, *Good Ware*, *Pause08*, and *Madebyoliver* from `www.flaticon.com`.

Vector graphics from Pressfoto, brgfx, macrovector and Jannoon028 on freepik.com
Other photos from `www.pixabay.com`.

I/O Lower Bound

Recall:

$$\mathcal{B} = \sum_{i=1}^{q+1} \Delta_i \log_2\left(\frac{N}{\Delta_i}\right) \qquad \text{with } \Delta_i = r_i - r_{i-1} \qquad (1 \leqslant i \leqslant q+1,\ r_0 = 0,\ r_{q+1} = N+1)$$

$$\boxed{\mathcal{B}_{I/O} = \sum_{i=1}^{q+1} \frac{\Delta_i}{B} \log_{\frac{M}{B}}\left(\frac{N}{\Delta_i}\right)} = \frac{\mathcal{B}}{B \log_2(M/B)} \underset{\text{(usually)}}{\ll} \frac{\mathcal{B}}{B}$$

Theorem (Lower bound)

External-memory multiple selection in expectation requires $\Omega(\mathcal{B}_{I/O}) - O\left(\frac{N}{B} \log_{M/B} B\right)$ *I/Os.*

Follows from general reduction (cmps bound $\rightsquigarrow$ I/Os bound)

Arge, Knudsen, Larsen: *A general lower bound on the I/O-complexity of comparison-based algorithms*, WADS **1993**

⚠ "finish-sorting" argument no longer rigorous

Recap: (Lazy) Funnelsort

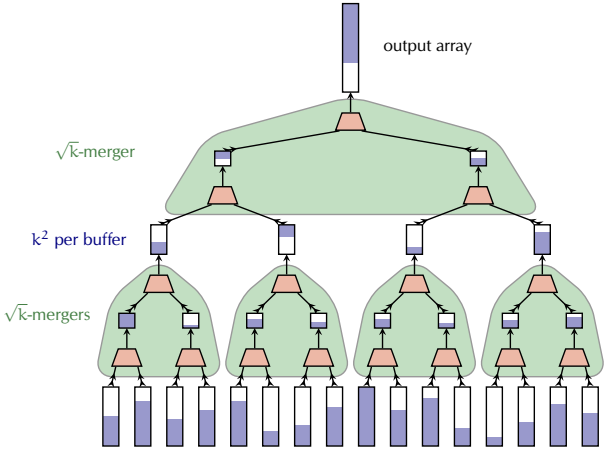
- **Funnelsort**
  - is a $k = \sqrt[4]{N}$-way Mergesort (*outer recursion*)
  - each realized by recursive binary merging (*inner recursion*)
  - with **judiciously sized buffers** for intermediate results (*funnel*)

Simplifying assumptions:

$N = 2^{2^i}$ and $d = 4$

(i.e., $\varepsilon \geqslant \frac{2}{3}$ in tall cache assumption)

$d > 2$ controls fanout ($\approx N^{1/d}$-way merging)



k-merger

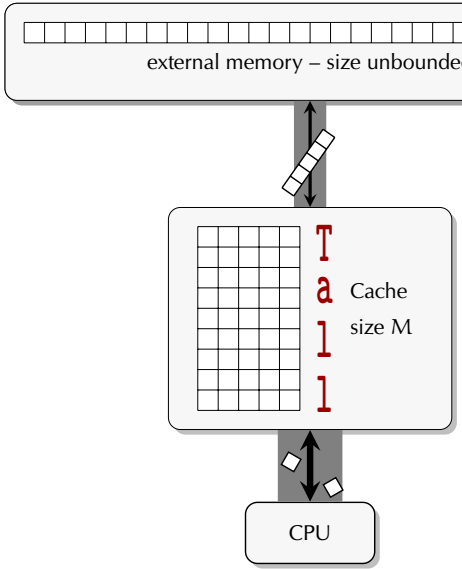
- Recursive structure (cf. van Emde Boas trees)
- largest buffers in middle layer

$\rightsquigarrow$ overall space $O(k^{5/2}) = O(N^{5/8})$

- Merge = **fill** output buffer
  - when input buffer empty, **recursively** fill it

$\approx$ ***I/O-optimal cache-oblivious gadget***
for $\hat{k}$-way merging with $\hat{k} \approx M^{0.3}$

Tall Caches



- **Tall Cache Assumption:** $\boxed{M \geqslant B^{1+\varepsilon}}$ think: $\varepsilon = 1$
  - $\rightsquigarrow$ $M/B \geqslant B^{\varepsilon}$
  - $\rightsquigarrow$ cache fits many cache lines
  - **necessary** for existence of I/O-optimal (comparison-based) cache-oblivious sorting algorithms

  Brodal, Fagerberg: *On the limits of cache-obliviousness*, STOC **2003**