Funnelselect: Cache-Oblivious Multiple Selection

Gerth Stølting Brodal
Aarhus University, Denmark

Sebastian Wild
University of Liverpool, UK

Abstract

We present the algorithm *funnelselect*, the first optimal randomized cache-oblivious algorithm for the multiple-selection problem. The algorithm takes as input an unsorted array of N elements and q query ranks $r_1 < \cdots < r_q$, and returns in sorted order the q input elements of rank $r_1, \ldots, r_q$, respectively. The algorithm uses expected and with high probability $O\big(\sum_{i=1}^{q+1} \frac{\Delta_i}{B} \cdot \log_{M/B} \frac{N}{\Delta_i} + \frac{N}{B}\big)$ I/Os, where B is the external memory block size, $M \geq B^{1+\varepsilon}$ is the internal memory size, for some constant $\varepsilon > 0$, and $\Delta_i = r_i - r_{i-1}$ (assuming $r_0 = 0$ and $r_{q+1} = N + 1$). This is the best possible I/O bound in the cache-oblivious and external memory models. The result is achieved by reversing the computation of the cache-oblivious sorting algorithm funnelsort by Frigo, Leiserson, Prokop and Ramachandran [FOCS 1999], using randomly selected pivots for distributing elements, and pruning computations that with high probability are not expected to contain any query ranks.

2012 ACM Subject Classification Theory of computation → Design and analysis of algorithms

Keywords and phrases Multiple selection, cache-oblivious algorithm, randomized algorithm, entropy bounds

Digital Object Identifier 10.4230/LIPIcs.ESA.2023.25

Funding *Gerth Stølting Brodal*: Independent Research Fund Denmark, grant 9131-00113B.

1 Introduction

We present the first optimal randomized cache-oblivious algorithm for the multiple-selection problem. Our result combines ideas from the cache-oblivious sorting algorithm funnelsort with existing multiple-selection algorithms. Many existing time- and comparison-optimal multiple-selection algorithms are already cache oblivious, but they are not optimal with respect to the number of I/Os performed when analyzed in the cache-oblivious model.

Let us start with a brief history of the multiple-selection problem. In 1961, Hoare presented the classic randomized sorting algorithm *quicksort*, published as Algorithm 64 in the Algorithms column of the Communications of the ACM [15]. Quicksort makes essential use of the randomized algorithm *partition* (Algorithm 63 [14]), that picks a random element, denoted a *pivot*, and partitions the elements into those smaller and larger than the pivot. By recursing on each subproblem, quicksort sorts an input of size N in expected $O(N \lg N)$ time and comparisons[1]. Hoare observed that if we are only interested in finding the rth smallest element in the input, denoted the element of *rank* r, we do not need to sort the input completely. By pruning recursive calls in quicksort not relevant for finding the rth smallest element, the resulting algorithm *find* (Algorithm 65 [16]) achieves expected $O(N)$ time. Chambers [7] generalized this idea to finding q elements of q given ranks $1 \leq r_1 < r_2 < \cdots < r_q \leq N$, in the following denoted the *multiple-selection* problem, by just skipping all recursive problems not containing any query rank. The expected running time is $O(N \lg q)$, but Prodinger [19] proved a tighter expected bound of $O(\mathcal{B} + N)$, where $\mathcal{B} = \sum_{i=1}^{q+1} \Delta_i \lg \frac{N}{\Delta_i}$ with $\Delta_i = r_i - r_{i-1}$, for $1 \leq i \leq q+1$, assuming $r_0 = 0$ and $r_{q+1} = N+1$.

[1] lg denotes the binary logarithm.

31st Annual European Symposium on Algorithms (ESA 2023).
Editors: Inge Li Gørtz, Martin Farach-Colton, Simon J. Puglisi, and Grzegorz Herman; Article No. 25; pp. 25:1–25:17
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

We call $\mathcal{B}$ the *entropy* of the multiple-selection query [3]. Dobkin and Munro [8] achieved matching asymptotic bounds for the worst-base time in the comparison model by using a deterministic linear-time (single) selection algorithm [4, 20] for the partitioning steps.

1.1 Model of Computation

In this paper we study the multiple-selection problem in a hierarchical-memory model, where we have an infinite external memory and an internal memory of capacity M elements, and where data is transferred between the internal and external memory in blocks of B consecutive elements. A block transfer is called an *I/O* (input/output operation). The I/O cost of an algorithm is the number of I/Os it performs. Aggarwal and Vitter [1] introduced this as the *external-memory model* and proved that sorting in this model requires $\Theta\big(\frac{N}{B}\log_{M/B}\frac{N}{B}\big)$ I/Os. The upper bound is, e.g., achieved by M/B-way mergesort and distributionsort algorithms, where the algorithms exploit knowledge of the parameters M and B.

Frigo *et al.* [12, 13] introduced the *cache-oblivious model*, that essentially is the same as the external-memory model, except that algorithms *do not know* M and B, and I/Os are assumed to be performed automatically by an optimal paging algorithm. As a consequence, cache-oblivious algorithms also adapt to multi-level memory hierarchies (under certain conditions [13]). The same paper introduced the cache-oblivious sorting algorithm *funnelsort* achieving the optimal external-memory I/O bound, assuming a *"tall-cache"*, $M = \Omega(B^2)$. Brodal and Fagerberg [6] observed that under the weaker tall-cache assumption $M \geq B^{1+\varepsilon}$, for a constant $\varepsilon > 0$, the optimal I/O bound increases by a factor $\Theta(1/\varepsilon)$.

Multiple selection was studied in external-memory by Hu *et al.* [17] and Barbay *et al.* [3]. The algorithms have an I/O cost of $O\big(\mathcal{B}_{\mathrm{I/O}} + \frac{N}{B}\big)$, where $\mathcal{B}_{\mathrm{I/O}} = \frac{\mathcal{B}}{B\lg(M/B)}$. A matching lower bound was sketched in [3] as a reduction from sorting, assuming the multiple-selection algorithm partitions the input elements into the gaps between the queried elements (most algorithms actually solve this problem, that Chambers denoted *partial sorting*). Hu *et al.* [17] considered the case where the queried elements can be returned in arbitrary order without partial sorting, and showed that without a tall-cache assumption, this problem can actually be solved asymptotically faster for a small number of queries q.

1.2 Results

Our first result is a lower bound for the external-memory multiple-selection problem (and not only for the partial-sorting problem as in the lower bound of Barbay *et al.* [3]).

▶ **Theorem 1** (Lower bound). *External-memory multiple selection in expectation requires* $\Omega(\mathcal{B}_{\mathrm{I/O}}) - O\big(\frac{N}{B}\log_{M/B} B\big)$ *I/Os.*

Note that an external-memory lower bound is also a cache-oblivious lower bound (for any online paging strategy), and that under a tall-cache assumption $M \geq B^{1+\varepsilon}$, for a constant $\varepsilon > 0$, the last term $O\big(\frac{N}{B}\log_{M/B} B\big) = O\big(\frac{1}{\varepsilon}\cdot\frac{N}{B}\big)$. The result is obtained by combining the comparison lower bound for the multiple-selection problem by Dobkin and Munro [8] with the general reduction technique of Arge *et al.* [2], that can derive an I/O-decision-tree lower bound from a comparison-decision-tree lower bound.

Our second result is the cache-oblivious algorithm funnelselect.

▶ **Theorem 2** (Funnelselect upper bound). *There exists a randomized cache-oblivious algorithm solving the multiple-selection problem using* $O\big(\mathcal{B}_{\mathrm{I/O}} + \frac{N}{B}\big)$ *I/Os in expectation and with high probability.*

Table 1 Algorithms for selection and multiple selection. CO = cache-oblivious, $\mathbb{E}$ = expected, wc = worst-case bounds. Note that Barbay *et al.* assume a tall cache, whereas Hu *et al.* do not.

Reference		**Comparisons**	**I/Os**	**Comments**
Single selection				
Hoare [16]	$\mathbb{E}$	$2\ln 2\mathcal{B} + 2N + o(N)$	$O(N/B)$	CO, randomized
Floyd & Rivest [11]	$\mathbb{E}$	$N + \min\{r, N-r\} + o(N)$	$O(N/B)$	CO, randomized
Blum *et al.* [4]	wc	$5.4305N$	$O(N/B)$	CO, deterministic
Schönhage *et al.* [20]	wc	$3N + o(N)$	?	deterministic, median
Dor & Zwick [9]	wc	$2.95 + o(N)N$	?	deterministic, median
Multiple selection				
Chambers [7, 19]	$\mathbb{E}$	$2\ln 2\mathcal{B} + O(N)$	$O((\mathcal{B} + N)/B)$	CO, randomized
Dobkin & Munro [8]	wc	$3\mathcal{B} + O(N)$	$O((\mathcal{B} + N)/B)$	CO, deterministic
Kaligosi *et al.* [18]	wc	$\mathcal{B} + o(\mathcal{B}) + O(N)$	$O((\mathcal{B} + N)/B)$	CO, deterministic
Hu *et al.* [17]	wc	$O(N \lg(q))$	$O(N/B \log_{M/B}(q/B))$	deterministic
	wc	$O(\mathcal{B} + N)$	$O(\mathcal{B}_{\mathrm{I/O}} + N/B)$	(from closer analysis)
Barbay *et al.* [3]	wc	$\mathcal{B} + o(\mathcal{B}) + O(N)$	$O(\mathcal{B}_{\mathrm{I/O}} + N/B)$	online, determ., $M \geq B^{1+\varepsilon}$
New (Theorem 2)	$\mathbb{E}$	$O(\mathcal{B} + N)$	$O(\mathcal{B}_{\mathrm{I/O}} + N/B)$	CO, randomized, $M \geq B^{1+\varepsilon}$

At the high level, the result is obtained by the standard approach of recursively partitioning by pivots and pruning computations not containing any query ranks. To achieve good I/O performance in the cache-oblivious model we pipeline the partitioning by essentially reversing the computations done by funnelsort, and replace each merging node by a partitioning node. Since we do not know the ranks of the pivots during the partitioning, we pick the pivots carefully from a random sample such that a concentration bound guarantees approximate ranks of the pivots, so we can truncate computations that with high probability do not contain any query ranks. Table 1 summarizes known and the new results.

1.3 Preliminaries and Notation

Throughout the paper we assume that the input to a multiple-selection algorithm are two arrays S and R, where S is an unsorted array of N elements from a totally ordered universe, and R is a sorted array $r_1, \ldots, r_q$ of q distinct query ranks, where $1 \leq r_1 < \cdots < r_q \leq N$. Our task is to report an array of the q order statistics $S_{(r_1)}, \ldots, S_{(r_q)}$, where $S_{(r)}$ is the rth smallest element in S, i.e., the element at index r in an array storing S after sorting it. If x is an element and S a set, we let $x < S$ denote that $x < y$ for all y in S. Unless stated otherwise, we assume that all elements in S are distinct.

1.4 Outline of Paper

In Section 2 we prove the I/O lower bound for multiple selection stated in Theorem 1. In Section 3 we present internal-memory and external-memory algorithms as a warm-up for the cache-oblivious algorithm in Section 4 achieving Theorem 2. In Section 5 we analyze the algorithm. In Section 6, we discuss how to extend the algorithm to partially sort the input, and in Section 7, we discuss how to deal with equal elements. Section 8 concludes with open problems.

2 Lower Bound

In this section we prove Theorem 1. Dobkin and Munro [8, Theorem 1] observed that the comparisons done by a comparison-based multiple-selection algorithm must classify the remaining elements into "*gaps*" between the selected elements, and by sorting each of these gaps with $\Delta_i - 1$ elements using $\Delta_i \lg \Delta_i - O(\Delta_i)$ additional comparisons, one can sort the input. Together with the $N \lg N - O(N)$ lower bound on comparison based sorting, we have

$$\#\text{comparisons for multiple selection} \; + \; \sum_{i=1}^{q+1} (\Delta_i \lg \Delta_i - O(\Delta_i)) \;\; \geq \;\; N \lg N - O(N) \; ,$$

implying a lower bound of $\mathcal{B} - O(N)$ on the number of comparisons for multiple selection, where $\mathcal{B} = \sum_{i=1}^{q+1} \Delta_i \lg(\frac{N}{\Delta_i})$. This holds for worst, average, and expected case.

To prove I/O lower bounds on external-memory algorithms, Arge *et al.* [2] presented a general reduction that converts a comparison lower bound into an I/O lower bound, by converting an I/O-decision tree T to a standard comparison decision tree T_c. An I/O-decision tree consists of unary I/O-nodes moving B elements between internal and external memory, and comparison nodes between two elements in internal memory. Their lower bound reduction [2, Corollary 5] relates for any input x, the number of I/Os in T, $\#\text{I/Os}_T(x)$, to the number of comparisons in T_c, $\#\text{comparisons}_{T_c}(x)$, as

$$\#\text{comparisons}_{T_c}(x) \;\; \leq \;\; N \lg B + \#\text{I/Os}_T(x) \cdot B\left(3 + \lg \frac{M-B}{B}\right) \; . \qquad (1)$$

Since the reduction relates comparisons and I/Os for each input instance, the reduction can be used to show worst-case, average-case, and expected-case lower bounds.

Plugging the $\mathcal{B} - O(N)$ comparison lower bound into eq. (1) we get

$$\sum_{i=1}^{q+1} \Delta_i \lg \frac{N}{\Delta_i} - O(N) \;\; \leq \;\; N \lg B + \#\text{I/Os} \cdot B\left(3 + \lg \frac{M-B}{B}\right) \; ,$$

implying the following I/O lower bound for multiple selection:

$$\#\text{I/Os} \;\; \geq \;\; \frac{1}{1 + \frac{3}{\lg(M/B)}} \cdot \mathcal{B}_{\text{I/O}} - O\left(\frac{N}{B} \log_{M/B} B\right) \;\; = \;\; \Omega(\mathcal{B}_{\text{I/O}}) - O\left(\frac{N}{B} \log_{M/B} B\right) \; ,$$

for $M \geq 2B$ and $\mathcal{B}_{\text{I/O}} = \sum_{i=1}^{q+1} \frac{\Delta_i}{B} \log_{M/B} \frac{N}{\Delta_i} = \frac{\mathcal{B}}{B \lg(M/B)}$. This concludes the proof of Theorem 1.

Aggarwal and Vitter [1, Theorem 3.1] proved that comparison-based external-memory sorting requires $\Omega(\frac{N}{B} \cdot \log_{M/B} \frac{N}{B})$ I/Os. This lower bound also applies to sorting in the cache-oblivious model. Brodal and Fagerberg [6, Corollary 2] showed that for a cache-oblivious sorting algorithm to be asymptotically optimal for all choices of M and B, a "tall-cache" assumption $M \geq B^{1+\varepsilon}$ is *necessary.* Since we can sort N elements using a multiple-selection algorithm by querying all ranks $1, \ldots, N$, a tall-cache assumption is also necessary for matching bounds for multiple selection in the cache-oblivious model.

3 Internal-Memory and External-Memory Multiple Selection

In this section we consider simple internal-memory and external-memory (cache-conscious) algorithms for multiple selection as a warm-up for our cache-oblivous algorithm in Section 4, which borrows ideas from both algorithms.

3.1 Internal Memory

A simple recursive internal-memory algorithm is MultiSelect (Algorithm 1). This is essentially Chamber's algorithm from 1971 [7], except for the choice of pivot. If there are no

Algorithm 1 Internal-memory multiple selection.

```
procedure MultiSelect($S[1..N]$, $R[1..q]$)
    if $R \neq \emptyset$ then
        $P \leftarrow$ median of $S$ (pivot)
        Partition $S$ into $S_1 < P < S_2$
        $\bar{r} \leftarrow |S_1| + 1$ (the rank of $P$ in $S$)
        Partition $R$ into $R_1 < \bar{r} < R_2$
        MultiSelect($S_1$, $R_1$)
        if $\bar{r} \in R$ then
            Report $P$
        MultiSelect($S_2$, $\{r - \bar{r} \mid r \in R_2\}$)
```

Algorithm 2 External-memory multiple selection (multi-way generalization of MultiSelect).

```
procedure MultiSelect$_{\mathrm{I/O}}$($S[1..N]$, $R[1..q]$)
    if $R \neq \emptyset$ then
        Find $\bar{k} - 1 \leq k - 1$ pivots $P_1 < \cdots < P_{\bar{k}-1}$ in $S$
        Partition $S$ into $S_1, \ldots, S_{\bar{k}}$ s. t. $P_{i-1} < S_i < P_i$ ($P_0 = -\infty$, $P_{\bar{k}} = +\infty$)
        $\bar{r}_i \leftarrow i + |S_1| + \cdots + |S_i|$ (the rank of $P_i$ in $S$)
        Partition $R$ into $R_1, \ldots, R_{\bar{k}}$ s. t. $\bar{r}_{i-1} < R_i < \bar{r}_i$ ($\bar{r}_0 = 0$, $\bar{r}_{\bar{k}} = N + 1$)
        for $i = 1, \ldots, \bar{k}$ do
            MultiSelect$_{\mathrm{I/O}}$($S_i$, $\{r - \bar{r}_{i-1} \mid r \in R_i\}$)
            if $\bar{r}_i \in R$ then
                Report $P_i$
```

query ranks in R, nothing needs to be reported. Otherwise, pick a pivot P from S, partition $S \setminus \{P\}$ into S_1 and S_2, such that $S_1 < P < S_2$, compute the rank $\bar{r}$ of the pivot P in S, partition $R \setminus \{\bar{r}\}$ into R_1 and R_2, such that $R_1 < \bar{r} < R_2$, and recurse on the subproblems (S_1, R_1) and (S_2, R_2). The pivot P is output before the second recursion if $\bar{r}$ is a query rank in R (so elements are reported in increasing rank order). This intuitively corresponds to a distributionsort/quicksort, where we truncate recursive calls not containing any query ranks in R.

In Algorithm 1, P is the exact median of S, but we could also have used an approximate median, or a randomly sampled pivot. Chamber's original algorithm uses a random element from S. Finding the pivot can be done using the deterministic linear-time median finding algorithms by Blum *et al.* [4] or the randomized algorithms by Hoare [16] or Floyd and Rivest [11]. Prodinger [19] proved that selecting a random pivot leads to expected overall $O(\mathcal{B} + N)$ time. Kaligosi *et al.* [18, Section 2] proved that Algorithm 1 achieves $O(\mathcal{B} + N)$ worst-case time, if a linear time median selection algorithm is used.

Algorithm MultiSelect is cache oblivious, since it is designed independently of the memory parameter B and M. All the above median algorithms are based on repeatedly scanning arrays and (analyzed in the cache-oblivious model) require $O(N/B)$ I/Os worst-case and expected, respectively. Since the additional work of MultiSelect can be implemented by repeatedly scanning arrays allocated on a stack, the I/O cost of the algorithm equals the internal computation time divided by the external-memory block size, i.e., $O(\mathcal{B}/B)$ I/Os. Our cache-oblivious algorithm from Section 4 improves upon this I/O cost by a factor $\Theta\big(\lg \frac{M}{B}\big)$.

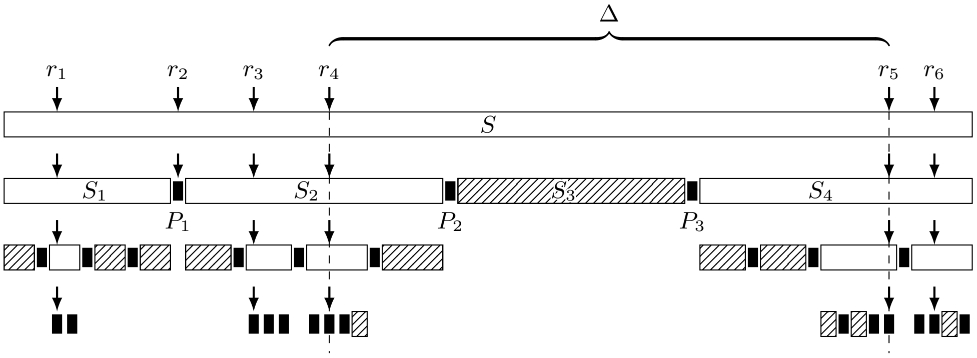

Figure 1 Recursion for $\textsc{MultiSelect}_{\mathrm{I/O}}$ with six query ranks and $k = 4$. Black squares are pivots, arrows show rank queries, and shaded areas are skipped subproblems with no query ranks.

3.2 External Memory

A generalization of MultiSelect better suited for external memory is to replace the binary partitioning by a multi-way partitioning. For a parameter $k \geq 2$, we assume the set S is partitioned into $\bar{k} \leq k$ subsets around $\bar{k} - 1$ pivots, where each set has size $O(|S|/k)$. The resulting algorithm is shown as $\textsc{MultiSelect}_{\mathrm{I/O}}$ in Algorithm 2. Figure 1 shows a recursion for $\textsc{MultiSelect}_{\mathrm{I/O}}$ on an example with six query ranks. If all sets S_i defined by the pivots have size at most $\alpha|S|$, where $1/k \leq \alpha < 1$, we denote the partitioning a (k, α)-*partitioning*. The algorithm MultiSelect (with exact medians) is the special case of $\textsc{MultiSelect}_{\mathrm{I/O}}$, where we use a $\left(2, \frac{1}{2}\right)$-partitioning.

There are several $(k, O(1/k))$-partitioning schemes described in the literature, e.g., a $(k, 1.5/k)$-partitioning method with $k = \sqrt{M/B}$ by Aggarwal and Vitter [1]. Here we describe a simpler $(k, 2/k)$-partitioning that incrementally inserts the N elements of S into buckets defined by a monotonically growing set of pivots, that also works for $k = \Theta(M/B)$. Initially there is one empty bucket and no pivot. Whenever a bucket reaches size $> 2N/k$ (i.e., the size is $1 + \lfloor 2N/k \rfloor$), the median of the bucket is selected as a new pivot, and the bucket is split around the pivot into two buckets with the elements smaller than and larger than the new pivot, respectively. Each new bucket has size at least $\lfloor N/k \rfloor$. Therefore, the total number of buckets created is at most k and each bucket contains at most $2N/k$ elements.

Crucial for the I/O effectiveness of this partitioning is that one memory block from each bucket is in memory while scanning S and distributing elements to buckets, i.e., $k \leq c\frac{M}{B}$ for a suitable constant $0 < c < 1$. Since each bucket can be split using $O(N/(kB))$ I/Os using the deterministic selection algorithm from [4], the total cost for creating a $(k, 2/k)$-partitioning of S is $O(N/B)$ I/Os, provided $k \leq c\frac{M}{B}$. A binary search to find the bucket for an element requires $\leq \lceil \lg(k-1) \rceil$ comparisons with pivots, i.e., in total $O(N \lg k)$ comparisons for distributing to buckets. Since each of the at most $k - 1$ bucket splits requires $O(N/k)$ comparisons [4], creating a $(k, 2k)$-partitioning requires $O(N \lg k)$ comparisons.

3.3 Analysis

We now analyze the comparison and I/O cost of $\textsc{MultiSelect}_{\mathrm{I/O}}$. Assume creating a (k, α)-partitioning has an (abstract) cost of $C \cdot N$, where $C = C(k, \alpha, M, B)$ does not depend on N. For example, when counting comparisons we have $C = \Theta(\lg k)$. The total cost of $\textsc{MultiSelect}_{\mathrm{I/O}}$ is the sum of the costs of all partitioning steps, i.e., C times the sum of the sizes of all the subsets partitioned by the algorithm (the white rectangles in Figure 1).

Consider any fixed gap Δ between two query ranks. We assume that the right query rank (but not the left) is part of the gap, so that all elements of S belong to exactly one gap. At each level of the recursion, the gap Δ intersects at most two subproblems that need to be partitioned, namely the subproblems containing the query ranks at the boundary of Δ (illustrated by the dashed lines in Figure 1). Since subproblems at depth d of the recursion have size at most $\alpha^d N$, the total cost we need to charge within gap Δ is at most C times

$$\sum_{d=0}^{\infty} \min(\Delta, 2\alpha^d N) \;\leq\; \Delta \left\lfloor \log_{1/\alpha} \frac{2N}{\Delta} \right\rfloor + \Delta \sum_{i=0}^{\infty} \alpha^i \;\leq\; \Delta \log_{1/\alpha} \frac{2N}{\Delta} + \frac{\Delta}{1-\alpha} \,.$$

Here we use that $2\alpha^d N$ is geometrically decreasing and $\Delta = 2\alpha^d N$ implies $d = \log_{1/\alpha} \frac{2N}{\Delta}$, i.e., in the sum, Δ is the term for the first $\lfloor \log_{1/\alpha} \frac{2N}{\Delta} \rfloor$ levels, whereas the remaining terms are geometrically decreasing, starting with at most Δ. Summing over all gaps we obtain total cost at most

$$C \cdot \left(\sum_{i=1}^{q+1} \Delta_i \log_{1/\alpha} \frac{2N}{\Delta_i} + \frac{N}{1-\alpha} \right) . \tag{2}$$

Recall that MultiSelect is the special case of MultiSelect$_{\mathrm{I/O}}$ with $k = 2$ and $\alpha = 1/2$. For comparisons, we have $C = O(1)$ per processed element in partitioning. By eq. (2), the total number of comparisons in MultiSelect is $O(\mathcal{B} + N)$. For MultiSelect$_{\mathrm{I/O}}$ we have $k = cM/B$ and $\alpha = 2/k$, and a cost of $C = O(1/B)$ I/Os per processed element, so by eq. (2), MultiSelect$_{\mathrm{I/O}}$ has a total cost of $O\big(\mathcal{B}_{\mathrm{I/O}} + \frac{N}{B}\big)$ I/Os. Alternatively, using the multiway partitioning method of Aggarwal and Vitter [1] with $k = \sqrt{M/B}$, $\alpha = 1.5/k$, and cost $C = O(1/B)$ for I/Os, we also get a total cost of $O\big(\mathcal{B}_{\mathrm{I/O}} + \frac{N}{B}\big)$ I/Os from eq. (2).

4 Cache-Oblivious Multiple Selection

In this section we present our cache-oblivious multiple-selection algorithm FunnelSelect (Algorithm 4). We first recall funnels for merging (Section 4.1) and then show that they can be used for partitioning (Section 4.2). FunnelSelect performs a single round of such a funnel-based partitioning, splitting the input into k parts of expected size $\Theta(N/k)$ using $k-1$ pivots, where $k = \Theta(N^{1/d})$ and $d = \max\{1 + 2/\varepsilon, 3\}$ under the tall-cache assumption $M \geq B^{1+\varepsilon}$. We then deal with each of the k parts with a non-empty set of rank queries by fully sorting it and returning the sought ranks.

However, to stay within the allowed I/O bound, we have to truncate partitioning, namely whenever neither side of the split is likely to contain a query rank (Section 4.4). To boost the probability of "guessing correctly" which buckets query ranks fall into, we also have to choose pivots judiciously (Section 4.3). Section 5 then proves Theorem 2.

4.1 Funnelsort

Since our cache-oblivious multiple-selection algorithm is heavily based on ideas from the optimal cache-oblivious sorting algorithm funnelsort by Frigo *et al.* [12, Section 4], we briefly recall funnelsort and in particular its k-merger construction here.[2] Funnelsort uses

[2] It should be noted that the cache-oblivious distributionsort algorithm in [12, Section 5] is a significantly different approach than the one taken by funnelsort and our algorithm, even though our algorithm highly resembles a classic internal-memory distributionsort algorithm.

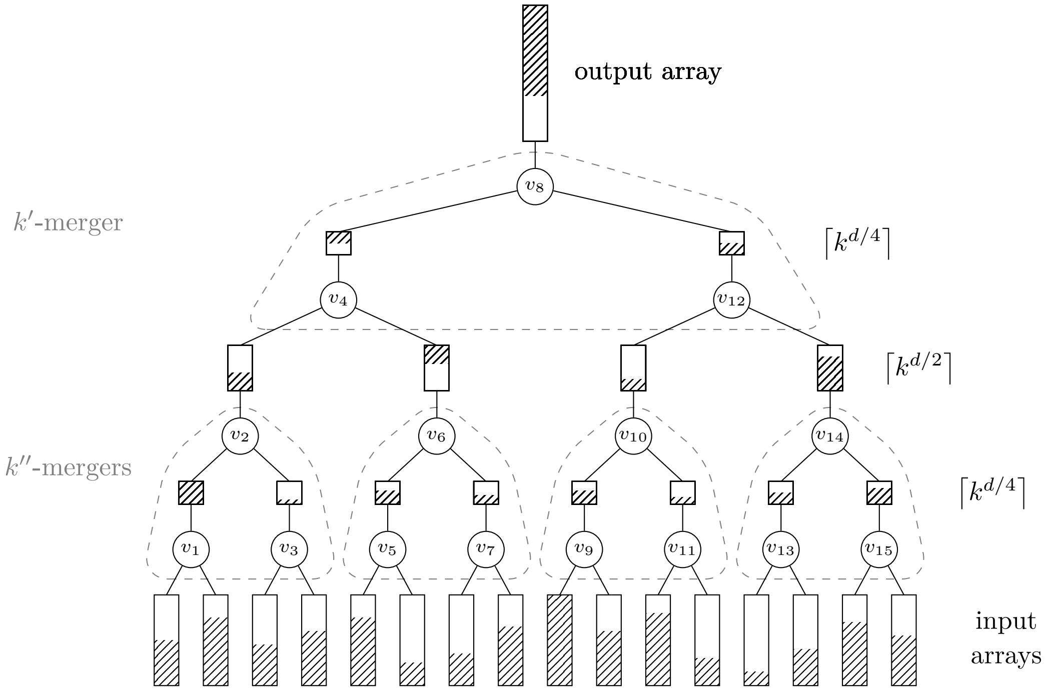

Figure 2 A k-merger in funnelsort for $k = 16$ input arrays. Content in the buffers is shaded; elements are added to the bottom of buffers and consumed from the top of buffers. The figure shows the situation where v_6 is in the process of filling its output buffer, after being recursively called from v_4 during its merging, which in turn has been called by v_8 during its merging. Buffer sizes for the three internal levels are shown next to the buffers.

$O(\frac{N}{B} \log_{M/B} \frac{N}{B})$ I/Os, assuming the tall-cache assumption $M \geq B^2$. Brodal and Fagerberg [5, Lemma 1] presented a lazy version of funnelsort, achieving the same I/O bound under the weaker tall-cache assumption $M \geq B^{1+\varepsilon}$, for any constant $\varepsilon > 0$. Funnelsort sorts an array of N elements by an outer recursion that partitions the input into k arrays each of size at most $\lceil N/k \rceil$, sorts these subarrays recursively, and then merges these arrays using a k-way construction named a *k-merger*. The parameter k depends on the tall-cache assumption (via ε) and the input size N: $k = 2^{\lceil \lg(N)/d \rceil} = \Theta(N^{1/d})$ for $d = 1 + 2/\varepsilon$.

A k-merger (see Figure 2) consists of a perfectly balanced binary tree of height $\lg k$ of binary *merger-nodes*, where each tree edge contains a buffer that is a sorted array of elements. Each merger-node consumes elements from two child buffers and feeds into its parent output buffer. When invoking a merger-node v, the node v fills its output buffer by merging the content of its input buffers until either the output buffer is full or one of the input buffers is exhausted. If the input buffer of a child w is exhausted, we recursively invoke w to fill its output buffer; then v continues to fill its output buffer.

The I/O efficiency of funnelsort hinges entirely on a judicious choice of buffer sizes. The buffers connecting the middle levels of binary mergers (between level $\lceil \lg(k)/2 \rceil$ and one below) can hold $\lceil k^{d/2} \rceil$ elements each. The construction is recursively applied to a $k' = 2^{\lceil \lg(k)/2 \rceil} \approx \sqrt{k}$-merger forming the top $\lceil \lg(k)/2 \rceil$ levels and the k' children each of which is a $k'' = 2^{\lfloor \lg(k)/2 \rfloor} \approx \sqrt{k}$-merger below the middle buffers; following a van-Emde-Boas layout of the binary tree and recursively allocating buffers consecutively in memory in that order.

4.2 Funnels for Partitioning

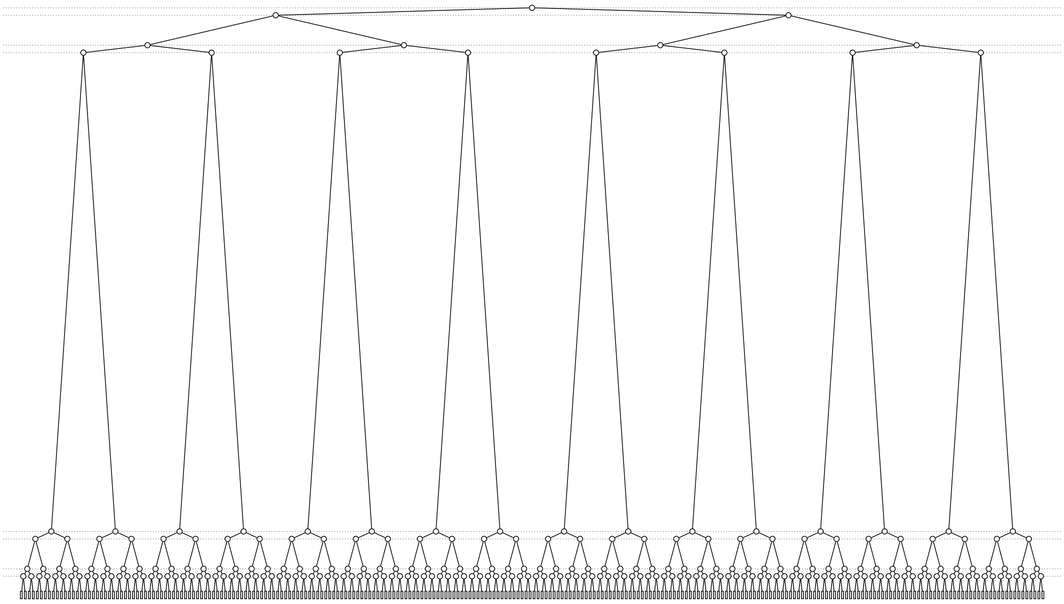

Figure 3 A 256-partitioner; splitter nodes are shown as circles (as in Figure 2); buffers are only shown as edges, but with the vertical length of the edges to scale with the buffer sizes for $d = 2$. (Buffer sizes by level are 4, 16, 4, 256, 4, 16, 4).

A key innovation of this paper is the *k-partitioner*, which uses funnels *in reverse*: instead of merging k runs, we push elements down the funnel while partitioning them around $k - 1$ pivots into k buckets. We use the same internal buffer sizes for a k-partitioner as in a k-merger; each buffer is organized as a queue and maintains an element count. The k output buffers at the bottom of a k-partitioner are conceptually unbounded (never full). Note that in partitioning, we always know the number N of elements and can allocate output buffers as linked lists of blocks of size $\Theta(N/k)$. Calling PARTITION (Algorithm 3) on a node v partitions elements around v's pivot and passes them to one of v's children, recursively emptying these whenever they become full. FUNNELPARTITION starts with all elements in the root's input buffer and then calls PARTITION on the root. After that, FLUSH recursively empties any remaining nonempty buffers.

Figure 2 can be read *mutatis mutandi* as a k-partitioner instead of a k-merger: Each node v_i stores a pivot P_i and PARTITION pushes elements towards the leaves. Buffers are consumed from the bottom and filled at the top of the hatched area. Figure 2 shows an overall PARTITION call at v_8, where first v_4 and then v_6 had run full; currently v_6 is moving elements from its parent buffer to its child buffers. Note that buffer sizes in Figure 2 are not drawn to scale; Figure 3 gives a more truthful representation.

The main property of k-partitioners is given in Lemma 3 below. It is similar to [6, Lemma 1], but we give a self-contained proof here.

▶ **Lemma 3** (Funnel lemma). *There exists a constant $c \geq 1$ so that the following holds. Let $d \geq 2$ be a constant. The size of a k-partitioner (excluding its output buffers) is bounded by $c \cdot k^{(d+1)/2}$. Assume $d \geq 2$ is such that $B^{1+\varepsilon} \leq M/3$ where $\varepsilon = 2/(d-1)$. Partitioning $N \geq k^d$ elements with FUNNELPARTITION around $k - 1$ pivots uses $N \lg(k)$ comparisons and incurs $O\big(\frac{N}{B}(\log_M(k) + 1) + k\big)$ I/Os.*

Algorithm 3 Operations on k-partitioners. The *full*() method returns whether a buffer has capacity for further elements. The *clear*() method removes all current elements from a buffer.

```
procedure FunnelPartition($S[1..N], P[1..k-1]$)
    Sort $P$
    Build $k$-partitioner, using $P$ as pivots (assigned in in-order to nodes)
    $r \leftarrow$ root node of $k$-partitioner
    Partition($r$, $S$)
    Flush($r$)
    Return $k$-partitioner output buffers

procedure Partition($v$, $S[1..N]$)
    for $x \in S$ do
        if $x \le v.pivot$ then
            if $v.leftBuffer.full()$ then
                Partition($v.left$, $v.leftBuffer$)
            $v.leftBuffer.append(x)$
        else
            if $v.rightBuffer.full()$ then
                Partition($v.right$, $v.rightBuffer$)
            $v.rightBuffer.append(x)$
    $S.clear()$

procedure Flush($v$)
    if $v \neq null$ then
        Partition($v.left$, $v.leftBuffer$)
        Flush($v.left$)
        Partition($v.right$, $v.rightBuffer$)
        Flush($v.right$)
```

Proof. Let $h = \lg(k)$; by definition of k, we have $h \in \mathbb{N}$. The space usage is given recursively by $s(k) = k' \cdot \lceil k^{d/2} \rceil + s(k') + k' \cdot s(k'')$; where $k' = 2^{\lceil h/2 \rceil}$ and $k'' = 2^{\lfloor h/2 \rfloor}$ are the number of leaves in the top funnel and the bottom funnels, respectively. Assuming $k = 2^{2^i}$ for $i \in \mathbb{N}$, this simplifies to $s(k) = k^{(d+1)/2} + (k^{1/2}+1)s(k^{1/2})$, which satisfies $s(k) \le S(k)$ where we set $S(k) = ck^{(d+1)/2}$ for a constant $c \ge 1$ that depends on initial conditions; if we use $s(4) = 2 \cdot 4^{d/2} + 3$ (space for the buffers and the 3 pivots), $c \ge 2.2$ suffices. The bound $s(k) \le S(k)$ indeed remains valid even when h is not a power of 2.

For the analysis of the I/O bound, let M and B with $B^{(d+1)/(d-1)} \le M/3$ ("tall-cache assumption") be given. We follow the recursive construction of the funnel until a $\hat{k}$-partitioner $\hat{F}$ satisfies $S(\hat{k}) \le M/3$, i.e., its it fits entirely into (a third of the) internal memory. For that choice, by the tall-cache assumption, the whole $\hat{k}$-partitioner and one block per child and parent buffer fit into internal memory: $S(\hat{k}) + (\hat{k}+1)B \le M$.

Call the edges/buffers connecting $\hat{F}$ to its parent and children *large* (if they exist). For the analysis, imagine removing all large edges; this leaves us with disconnected *base trees*, which in the k-partitioner are connected only by the large edges. Note that between any two levels, either none or all edges are large. However, unless $k = 2^{2^i}$, the height $\hat{h} = \lg(\hat{k})$ of a base tree can vary between $\underline{h} = \lg(M/(3c))/(d+1)$ and $\overline{h} = 2\underline{h}$.

Now consider a call to Partition at the root of a base tree $\hat{F}$ with $\hat{k}$ leaves. Over the course of (recursively) pushing elements down through $\hat{F}$, we incur I/Os for loading the buffers and pivots of $\hat{F}$. Since base trees fit entirely into internal memory, unless there is another call triggered on a child base tree upon a full (large) buffer, we will only load buffers inside the $\hat{k}$-partitioner into memory once, at a total cost of $O(S(\hat{k})/B)$ I/Os; we also need to bring one block for each parent and child buffer into memory, using $O(\hat{k})$ I/Os. We now distinguish two cases.

Case (1): If $\hat{k} = k$, i.e., $\hat{F}$ is the entire k-partitioner. Since $\hat{F}$ (as well as one block per input and output buffer) fits into internal memory, we only need to load it once, at a cost of $O(S(k)/B + k)$ I/Os. This is $O(k^{(d+1)/2}/B + k) = O(k^d/B + k) = O(N/B + k) = O(N/B(\log_M(k) + 1) + k)$ as claimed.

Case (2): Otherwise, $\hat{k} < k$. We will argue that the following potential scheme pays for all I/Os costs: Whenever an element is inserted into a large buffer, it releases $\Theta(1/B)$ potential.

We first show that charging all elements for this released potential yields the desired bound. Between any two large buffers, one partitioning phase moves an element at least $\underline{h}$ levels down the tree. Overall, the N elements need to travel at most $h = \lg(k)$ levels down, hence each element can be inserted at most $\lceil h/\underline{h} \rceil = O(\log_M(k)+1)$ times into a large buffer, giving an overall potential of $O\big(\frac{N}{B}(\log_M(k) + 1)\big) = O\big(\frac{N}{B}(\log_M(k) + 1) + k\big)$ as claimed.

It remains to prove that the released potential exceeds the incurred I/O cost. First observe that we initially have to load each $\hat{F}$ when it is first used; likewise, we have to potentially load each $\hat{F}$ an additional time during the final Flush calls. These two load events sum to $O(S(k)/B + k)$ I/Os, which is $O(N/B + k)$ (as in case (1) above). Additionally, at any point in time during a Partition call on $\hat{F}$, we can get recursive Partition calls on $\hat{F}$'s child base trees in case their buffer becomes full; such a recursive call can evict $\hat{F}$ from the internal memory, and it has to be loaded before Partition resumes on $\hat{F}$, causing additional I/Os. We cannot say *when* these evictions will happen, but every eviction of $\hat{F}$ implies that a batch of β elements have been pushed down from $\hat{F}$'s input buffer to a child's buffer, where β is the size of the buffers below $\hat{F}$. By the recursive funnel construction, $\beta = \Omega(\hat{k}^d)$, so we must have seen a total release of $\Theta(\beta/B) = \Omega(\hat{k}^d/B)$ in potential. Since $\hat{k}$ is the first value with $S(\hat{k}) \le M/3$, we have $S(2\hat{k}^2) > M/3$, which implies, $M = O(\hat{k}^{d+1})$. By the tall-cache assumption, this implies $B = O(M^{(d-1)/(d+1)}) = O\big((\hat{k}^{(d+1)})^{(d-1)/(d+1)}\big) = O(\hat{k}^{d-1})$, so $\hat{k} = O(\hat{k}^d/B)$. Hence, the cost of loading $\hat{F}$ again after an eviction of $O(S(\hat{k})/B + \hat{k}) = O(\hat{k}^d/B)$ I/Os is covered by a release in potential. ◀

On a conceptual level, a k-partitioner can be thought of as a cache-oblivious gadget for repeatedly partitioning with (variable) fan-out in $\Omega(M^{1/(d+1)}) \cap O(M^{2/(d+1)})$ that, when applied to $\Omega(M^{d/(d+1)})$ elements, uses $O(1/B)$ I/Os per element and partitioning round. This results from the bounds on $\hat{k}$ in base trees in the proof above.

4.3 Selecting Pivots

For funnelselect, we choose pivots $P_1, \ldots, P_{k-1}$ as follows; see also Figure 4: We first sample each element in the input S with probability $p = 1/\lg N$. The resulting sample $\bar{S}$ is sorted. Finally we select the pivots as the ideal pivots in the sample, using the (expected) sample size pN: the ith pivot P_i is the $\lfloor 1 + ipN/k \rfloor$th smallest element of $\bar{S}$. If $\bar{S}$ is too small, i.e., if $|\bar{S}| < \lfloor (k-1)pN/k \rfloor$ or $|\bar{S}| > 2pN$, we declare the pivot selection *failed* and repeat the sampling process.

In Lemma 6 we prove that all pivots are expected "close" to the ideal pivots in S. We call

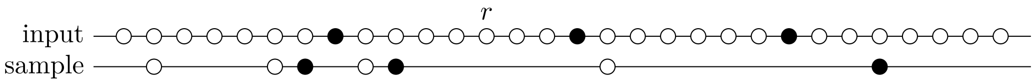

Figure 4 Using sampling to select $k-1$ pivots for a k-way partitioning of the input, when $k=4$ and $N=30$. The expected sample size was 9 elements, but only 7 were actually sampled. Top shows the sorted input and $k-1$ ideal pivots, whereas the bottom shows similarly $k-1$ ideal pivots for a sample of the input points. Note the input of rank r is in the 2nd block of white nodes defined by the ideal pivots, but would be in the 3rd block defined by the actual pivots from the sample.

the ith pivot P_i "ξ-bad" if its rank is more than ξ away from its ideal rank, more formally

$$P_i \text{ is "}\xi\text{-bad" if and only if } \big|S \cap (-\infty, P_i]\big| \;\notin\; \big[\lfloor 1+iN/k\rfloor \,-\, \xi\,,\, \lfloor 1+iN/k\rfloor \,+\, \xi\big]\,.$$

We will call P_i *bad* if it is ξ-bad; otherwise it is *good*. We call a pivot selection "bad" if any of the pivots is bad, and "good" otherwise. We select $\xi = \lceil N^{1/2+\delta}\rceil$ for a small constant δ, where $0 < \delta < 1/6$. Since we assume $d \ge 3$, $k = \Theta\big(N^{1/d}\big)$ implies $N/k = \Omega\big(N^{2/3}\big)$, and $\xi < \frac{1}{2}N/k$ for sufficiently large N. We get the following fact:

▶ **Fact 4** (Good pivots). *If $\xi < \frac{1}{2}N/k$ and all pivots are good,*

(a) *no bucket S_i contains more than $N/k + 2\xi \le 2N/k$ elements, and*

(b) *a query for a rank r will fall into one of at most two buckets: the $\lceil (r-\xi)/N \cdot k\rceil$th or the $\lceil (r+\xi)/N\cdot k\rceil$th bucket.*

Our analysis makes iterated use of the following Chernoff bound.

▶ **Lemma 5** ([10, Theorem 1.1]). *If $X_1, \ldots, X_n$ are independent random variables in $[0,1]$, and $X = X_1 + \cdots + X_n$, then for all $t > 0$ we have*

$$\Pr[X < \mathbb{E}[X] - t],\ \Pr[X > \mathbb{E}[X] + t] \;\;\le\;\; e^{-2t^2/n}\,.$$

▶ **Lemma 6.** *The probability that P_i is ξ-bad is bounded by $2\exp\big(-2\xi^2p^2/N\big)$.*

Proof. P_i is bad if its rank in S is too small ("small bad") or too big ("big bad"). We first consider too small ranks. By choice, there are ipN/k elements in $\bar{S}$ smaller than P_i; if P_i has small-bad rank, all of these elements must be of rank $< iN/k - \xi$ in S. That means, from these $iN/k - \xi - 1$ smallest elements in S, we have chosen at least ipN/k into $\bar{S}$. Since each choice is done independently with probability $p = 1/\lg N$, the number chosen for the sample is $X \overset{\mathcal{D}}{=} \mathrm{Bin}(iN/k - \xi - 1, p)$, a random variable with binomial distribution and expectation $\mathbb{E}[X] = p(iN/k - \xi - 1)$. We have

$$\begin{aligned}\mathbb{P}[P_i \text{ "small bad"}] \;&\le\; \mathbb{P}[X \ge piN/k] \;\le\; \mathbb{P}[X > \mathbb{E}[X] + p\xi] \\ &\underset{\text{Lemma 5}}{\le} \exp\!\left(-2\frac{(p\xi)^2}{iN/k - \xi - 1}\right) \;\le\; \exp\!\left(-\frac{2\xi^2p^2}{N}\right).\end{aligned}$$

For the "big-bad" case, we must have chosen at most ipN/k elements from the $iN/k + \xi + 1$ smallest elements in S into $\bar{S}$. With $X' \overset{\mathcal{D}}{=} \mathrm{Bin}(iN/k + \xi + 1, p)$, we obtain

$$\mathbb{P}[P_i \text{ "big bad"}] \;\le\; \mathbb{P}[X' \le piN/k] \;\le\; \mathbb{P}[X' < \mathbb{E}[X'] - p\xi] \;\le\; \exp\!\left(-\frac{2\xi^2p^2}{N}\right).$$

By the union bound, P_i is bad with probability at most $2\exp\big(-2\xi^2p^2/N\big)$. ◀

▶ **Lemma 7.** *The probability that the sample is too small to choose $k-1$ pivots is bounded by* $\exp\bigl(-2(pN/k-1)^2/N\bigr)$.

Proof. For the sample to be too small, we must have selected at most $(k-1)pN/k$ elements into the sample. Since $|\bar{S}| \overset{\mathcal{D}}{=} \mathrm{Bin}(N,p)$, we have by Lemma 5

$$\mathbb{P}[\bar{S} \text{ too small}] \quad = \quad \mathbb{P}[|\bar{S}| \le \mathbb{E}[|\bar{S}|] - pN/k] \quad \le \quad \exp\left(-2\frac{(pN/k-1)^2}{N}\right) .$$

◀

▶ **Corollary 8.** *With high probability, the pivot choice is well-defined and all $k-1$ pivots of one sampling round are good.*

Proof. By Lemmas 6 and 7 and the union bound, the probability that the sample is too small to choose our pivots or that any P_i is bad is at most

$$2(k-1)\exp\bigl(-2\xi^2p^2/N\bigr) + \exp\bigl(-2(pN/k-1)^2/N\bigr) \quad \le \quad 2k\exp\bigl(-\Omega\bigl(N^{2\delta}/\lg^2 N\bigr)\bigr) .$$

The inequality follows from $p = 1/\lg(N)$, $\xi = \Omega\bigl(N^{1/2+\delta}\bigr)$, $N/k = \Omega\bigl(N^{2/3}\bigr)$, and $\delta < 1/6$. This probability tends to 0 with speed superpolynomial in N. ◀

4.4 Truncated Partitioning

The algorithms from Section 3 achieve optimal cost from simply not recursing on subproblems not containing any query rank; after partitioning, it is obvious which subproblems are "query free". In a cache-oblivious algorithm, we have to truncate partitioning *inside* the k-partitioner.

After k-partitioning the input, elements are split into k buckets; let us denote these buckets by $S_1, \ldots, S_k$. By Fact 4(b), when pivots are good, a query rank r will fall in one of two buckets: one in $S(r) = \{S_{\lceil (r-\xi)/N \cdot k \rceil}, S_{\lceil (r+\xi)/N \cdot k \rceil}\}$. Buckets in the set $QF = \{S_1, \ldots, S_k\} \setminus \bigcup_{r \in R} S(r)$ do not contain any query ranks whenever pivots are good; we call these buckets *"expected query-free"*. Note that this is a property solely of R and k and hence QF can be determined by scanning R before partitioning commences.

When constructing the k-partitioner F, we check in a depth-first traversal whether all leaves below a binary partitioning node v are in QF; if so, we remove v and rewire its parent to send elements directly to an output buffer instead of v's input buffer. The sizes of buffers between partitioning nodes and the PARTITION methods remain unchanged. By generating the output buffers for the leaves of F consecutive in memory, before all internal buffers and reserving $2N/k$ space for each, this truncation operation simply changes one pointer.

4.5 Funnelselect

The overall algorithm FUNNELSELECT is shown in Algorithm 4. It applies one round of truncated k-partitioning as described above. For each of the resulting buckets, we then simply invoke an existing I/O-optimal cache-oblivious sorting algorithm and report the sought ranks. This can be done as a stack-based computation, so that no extra I/Os are paid for reporting elements, but instead they are reported while they are in main memory anyways from sorting.

5 Analysis

▶ **Theorem 9.** *Algorithm FUNNELSELECT is cache oblivious and uses $O\bigl(\mathcal{B}_{\mathrm{I/O}} + \frac{N}{B}\bigr)$ I/Os to report q query ranks $r_1 < \cdots < r_q$ from an unsorted array of N elements. With high probability it does not fail.*

Algorithm 4 Our overall cache-oblivious multiple-selection algorithm.

```
procedure FunnelSelect($S[1..N]$, $R[1..q]$, $\delta$)
    $p \leftarrow 1/\lg N$
    $k \leftarrow 2^{\lceil \lg(N)/d \rceil}$
    $\xi \leftarrow \lceil N^{1/2+\delta} \rceil$
    Scan $S$, copy each element into the sample $\bar{S}$ i.i.d. with prob. $p$
    if $|\bar{S}| \le (k-1)pN/k \,\vee\, |\bar{S}| < \frac{1}{2}Np \,\vee\, |\bar{S}| > 2Np$ then
        return Fail
    Sort $\bar{S}$
    for $i = 1, \dots, k-1$ do
        $P_i \leftarrow \bar{S}[\lfloor 1 + ipN/k \rfloor]$ (select pivots from $\bar{S}$)
    Construct $k$-partitioner $F$ using pivots $P_1, \dots, P_{k-1}$; let $S_1, \dots, S_k$ be its leaf buckets
    for $r \in R$ do
        $b_1 \leftarrow \lceil (r - \xi)/N \cdot k \rceil$ and $b_2 \leftarrow \lceil (r + \xi)/N \cdot k \rceil$
        Mark leaf buckets $S_{b_1}$ and $S_{b_2}$ as *expected query free*
    for node $v$ in bottom-up traversal of $F$ do
        if both of $v$'s children are marked *expected query free* then
            Mark $v$ as *expected query free*
            Delete $v$'s children
    for node $v$ in preorder traversal of $F$ do
        if $v$ marked *expected query free* then
            Declare $v$'s parent an *expected query free* output buffer
            Delete $v$
    Partition($F.root$, $S$)
    Flush($F.root$)
    $L_1, \dots, L_{\hat{k}} \leftarrow$ leaf buckets in $F$
    $\ell \leftarrow 0$
    for $i = 1, \dots, \hat{k}$ do
        $R' \leftarrow R \cap (\ell, \ell + |L_i|]$
        if $R' \neq \emptyset$ then
            if $L_i$ marked *expected query free* $\vee\, |L_i| > 2N/k$ then
                return Fail
            else
                Sort $L_i$ using an I/O-optimal cache-oblivious sorting algorithm.
                for $r \in R'$ do
                    Report $L_i[r - \ell]$
        $\ell \leftarrow \ell + |L_i|$
```

Proof. Let us first deal with failures. We let the algorithm fail if $|\bar{S}|$ is smaller than $\frac{1}{2}Np$ or larger than $2Np$. From Lemma 5, with high probability, this does not happen. The only other cause for failure are bad pivots; by Corollary 8 with high probability, this also does not happen.

For the I/O cost, we consider the different steps in turn. The I/O cost for computing the pivots consists of $O(N/B)$ I/Os to scan the input to construct the sample $\bar{S}$ of size at most $\bar{N} = 2N/\lg N$. To sort the sample, we can use standard top-down mergesort, yielding $O(\bar{N}/B \cdot \lg(\bar{N}/M)) = O(N/B)$ I/Os for sorting $\bar{S}$ and $O(N/B)$ I/Os to extract the pivots from $\bar{S}$. In total selecting the pivots requires $O(N/B)$ I/Os.

The buffers of F can be built sequentially, using $O(k^{(d+1)/2}/B) = O(N^{(d+1)/2d}/B) = O(N/B)$ I/Os (Lemma 3). Preparing the leaf buffers additionally touches $k = O(N^{1/d})$ positions; if $N^{1/d} > N/B$, then $B > N^{(d-1)/d}$ and by the tall-cache assumption, $M \ge B^{(d+1)/(d-1)} > N^{(d+1)/d}$, so the entire input fits in internal memory and the k accesses are cached. The same applies for truncating the k-partitioner; anything touching $O(k)$ random positions incurs $O(N/B)$ I/Os. Marking leaves and nodes as expected query-free can be done by scanning R (which is sorted), hence we use $O(q/B) = O(N/B)$ I/Os.

The key step is the k-partitioner. As in the proof of Lemma 3, we define $\hat{k}$ as the level in the recursive construction of the partitioner where the $\hat{k}$-partitioner first fits into internal memory. We overestimate the actual cost by always assuming the smallest $\hat{k} = (M/3c)^{1/(d+1)}$. As shown there, the k-partitioner has up to constant factors the same I/O cost as a repeated $\hat{k}$-way external-memory partitioning: $O(1/B)$ I/Os per element and $\hat{k}$-way split. This remains true when truncating the same subtrees in both algorithms. We can hence bound the cost of funnel partitioning as in Section 3 by charging the lengths of segments that are split further (white rectangles in Figure 1) to individual gaps Δ. For MultiSelect$_{\mathrm{I/O}}$, charging a gap Δ on each level either Δ or the 2 segments containing its endpoints, whichever is less, was sufficient to cover all partitioning costs. For FunnelSelect, due to marking (up to) two leaf buckets as not query-free for the endpoints, we can have up to 4 segments on any level that still require partitioning. The rest of the analysis is the same, though, and with $C = O(1/B)$, $k = \hat{k}$, and $\alpha = 2/\hat{k}$, we obtain an upper bound of

$$C \cdot \left(\sum_{i=1}^{q+1} \Delta_i \log_{1/\alpha} \frac{4N}{\Delta_i} + \frac{N}{1-\alpha} \right) \quad = \quad O(\mathcal{B}_{\mathrm{I/O}} + N/B)$$

I/Os for partitioning.

The last part of the algorithm, solving subinstances of multiple selection within leaf buckets, could be solved recursively, but as we now show, fully sorting such buckets also fits our desired I/O bound. This improves the failure probability as sorting can be deterministic. A subproblem L_i to recurse on is never declared query-free and hence moves all the $\lg k$ levels down the k-partitioner. For $N' = |L_i|$, L_i hence contributes $\Theta\big(\frac{N'}{B} \log_M k\big) = \Theta\big(\frac{N'}{B} \log_{M/B} N\big)$ I/Os to the partitioning cost, since $k = \Theta\big(N^{1/d}\big)$ and under our tall-cache assumption $\lg \frac{M}{B} = \Theta(\lg M)$. This is an upper bound on the I/O cost of sorting N' elements. Any I/O-efficient cache-oblivious sorting method (such as funnelsort) hence suffices for overall $O(\mathcal{B}_{\mathrm{I/O}} + \frac{N}{B})$ I/Os. ◀

▶ **Corollary 10.** *There exists a randomized cache-oblivious algorithm solving the multiple-selection problem using expected and with high probability $O\big(\mathcal{B}_{\mathrm{I/O}} + \frac{N}{B}\big)$ I/Os.*

Proof. FunnelSelect is formulated as a Monte-Carlo algorithm with worst-case time matching our expected-case time, but which can Fail occasionally. Repeating any failed execution turns it into a Las-Vegas algorithm with $O\big(\mathcal{B}_{\mathrm{I/O}} + \frac{N}{B}\big)$ expected I/Os; since the failure probability is superpolynomially small, we obtain the same bound with high probability. ◀

6 Partial Sorting

In internal memory, multiple selection would usually rearrange the input in place so that after the call to the multiple-selection algorithm, the sought elements are at indices $r_1, \ldots, r_q$. One would then not even return these elements explicitly. In external memory, this variant is less desirable, as one would have to pay q I/Os for accessing the elements by index later. Hence we defined the multiple-selection problem to return the elements of given ranks. However, it can be useful to also obtain the input partitioned around these returned values. Since all our multiple-selection algorithms conceptually follow an inorder traversal of a recursion tree and report sought elements (in sorted order) when they are identified, it is easy to augment the algorithms to produce a partitioned copy of the input array along the way. For funnelselect, we just have to make sure that output buckets are allocated sequentially in memory from left to right. That way we can output all elements falling between two returned pivots.

7 Allowing Identical Elements

In the previous sections we assumed elements to be distinct. A generic way to allow identical elements in the algorithms is by letting the algorithms process pairs (x, i), where x is the ith element in the array S, and break comparison ties between identical elements by comparing by their input position. This ensures all elements are considered distinct.

The drawback is that the computations need to process and store all these input positions. To avoid this overhead, one needs to address the problem directly by the individual algorithms. In the algorithms one needs to handle that multiple elements can be equal to the pivots. In the partitioning steps one needs to keep track of the number of elements equal to the pivots and only partition the elements not equal to pivots. Finally, one need to use this information gathered to handle that a pivot can span a range of ranks and be the answer to multiple query ranks.

8 Conclusion and Open Problems

We presented the first cache-oblivious multiple-selection algorithm that achieves the optimal I/O cost even when taking the (entropy of the) ranks to select into account.

A natural open problem is to find a *deterministic* cache-oblivious multiple-selection algorithm that achieves the same I/O bound as our randomized algorithm. Another interesting direction to explore is the "online" version of multiple selection studied in [3], where the ranks are given one after the other in arbitrary order and the algorithm has to produce the element of a given rank before the next rank is revealed. Investigating the practical performance of funnelselect is another route to pursue.

References

1 Alok Aggarwal and Jeffrey Scott Vitter. The input/output complexity of sorting and related problems. *Commun. ACM*, 31(9):1116–1127, 1988. doi:10.1145/48529.48535.

2 Lars Arge, Mikael B. Knudsen, and Kirsten Larsen. A general lower bound on the I/O-complexity of comparison-based algorithms. In Frank K. H. A. Dehne, Jörg-Rüdiger Sack, Nicola Santoro, and Sue Whitesides, editors, *Algorithms and Data Structures, Third Workshop, WADS '93, Montréal, Canada, August 11-13, 1993, Proceedings*, volume 709 of *Lecture Notes in Computer Science*, pages 83–94. Springer, 1993. doi:10.1007/3-540-57155-8_238.

3 Jérémy Barbay, Ankur Gupta, Srinivasa Rao Satti, and Jon Sorenson. Near-optimal online multiselection in internal and external memory. *Journal of Discrete Algorithms*, 36:3–17, jan 2016. doi:10.1016/j.jda.2015.11.001.

4 Manuel Blum, Robert W. Floyd, Vaughan R. Pratt, Ronald L. Rivest, and Robert Endre Tarjan. Time bounds for selection. *J. Comput. Syst. Sci.*, 7(4):448–461, 1973. doi:10.1016/S0022-0000(73)80033-9.

5 Gerth Stølting Brodal and Rolf Fagerberg. Cache oblivious distribution sweeping. In Peter Widmayer, Francisco Triguero Ruiz, Rafael Morales Bueno, Matthew Hennessy, Stephan J. Eidenbenz, and Ricardo Conejo, editors, *Automata, Languages and Programming, 29th International Colloquium, ICALP 2002, Malaga, Spain, July 8-13, 2002, Proceedings*, volume 2380 of *Lecture Notes in Computer Science*, pages 426–438. Springer, 2002. doi:10.1007/3-540-45465-9_37.

6 Gerth Stølting Brodal and Rolf Fagerberg. On the limits of cache-obliviousness. In Lawrence L. Larmore and Michel X. Goemans, editors, *Proceedings of the 35th Annual ACM Symposium on Theory of Computing, June 9-11, 2003, San Diego, CA, USA*, pages 307–315. ACM, 2003. doi:10.1145/780542.780589.

7 J. M. Chambers. Partial sorting [M1] (algorithm 410). *Commun. ACM*, 14(5):357–358, 1971. doi:10.1145/362588.362602.

8 David P. Dobkin and J. Ian Munro. Optimal time minimal space selection algorithms. *J. ACM*, 28(3):454–461, 1981. doi:10.1145/322261.322264.

9 Dorit Dor and Uri Zwick. Selecting the median. *SIAM Journal on Computing*, 28(5):1722–1758, 1999. doi:10.1137/s0097539795288611.

10 Devdatt P. Dubhashi and Alessandro Panconesi. *Concentration of Measure for the Analysis of Randomized Algorithms*. Cambridge University Press, 2009. URL: http://www.cambridge.org/gb/knowledge/isbn/item2327542/.

11 Robert W. Floyd and Ronald L. Rivest. Expected time bounds for selection. *Communications of the ACM*, 18(3):165–172, March 1975. doi:10.1145/360680.360691.

12 Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. In *40th Annual Symposium on Foundations of Computer Science, FOCS '99, 17-18 October, 1999, New York, NY, USA*, pages 285–298. IEEE Computer Society, 1999. doi:10.1109/SFFCS.1999.814600.

13 Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. *ACM Trans. Algorithms*, 8(1):4:1–4:22, 2012. doi:10.1145/2071379.2071383.

14 C. A. R. Hoare. Algorithm 63: partition. *Commun. ACM*, 4(7):321, 1961. doi:10.1145/366622.366642.

15 C. A. R. Hoare. Algorithm 64: quicksort. *Commun. ACM*, 4(7):321, 1961. doi:10.1145/366622.366644.

16 C. A. R. Hoare. Algorithm 65: find. *Commun. ACM*, 4(7):321–322, 1961. doi:10.1145/366622.366647.

17 Xiaocheng Hu, Yufei Tao, Yi Yang, and Shuigeng Zhou. Finding approximate partitions and splitters in external memory. In *Proceedings of the 26th ACM symposium on Parallelism in algorithms and architectures*. ACM, June 2014. doi:10.1145/2612669.2612691.

18 Kanela Kaligosi, Kurt Mehlhorn, J. Ian Munro, and Peter Sanders. Towards optimal multiple selection. In Luís Caires, Giuseppe F. Italiano, Luís Monteiro, Catuscia Palamidessi, and Moti Yung, editors, *Automata, Languages and Programming, 32nd International Colloquium, ICALP 2005, Lisbon, Portugal, July 11-15, 2005, Proceedings*, volume 3580 of *Lecture Notes in Computer Science*, pages 103–114. Springer, 2005. doi:10.1007/11523468_9.

19 Helmut Prodinger. Multiple Quickselect – Hoare's Find algorithm for several elements. *Information Processing Letters*, 56(3):123–129, November 1995. doi:10.1016/0020-0190(95)00150-b.

20 Arnold Schönhage, Mike Paterson, and Nicholas Pippenger. Finding the median. *J. Comput. Syst. Sci.*, 13(2):184–199, 1976. doi:10.1016/S0022-0000(76)80029-3.