Cache-oblivious comparison-based algorithms on multisets

Arash Farzan[1], Paolo Ferragina[2], Gianni Franceschini[2], and J. Ian Munro[1]

[1] {afarzan, imunro}@uwaterloo.ca
School of Computer Science, University of Waterloo
[2] {ferragin,francesc}@di.unipi.it
Department of Computer Science, University of Pisa

Abstract. We study three comparison-based problems related to multisets in the cache-oblivious model: Duplicate elimination, multisorting and finding the most frequent element (the mode). We are interested in minimizing the cache complexity (or number of cache misses) of algorithms for these problems in the context under which cache size and block size are unknown. We give algorithms with cache complexities that are within a constant factor of the optimal for all the first two problems. For the mode, The deterministic algorithm differs from the lower bound by a sublogarithmic factor. We can achieve optimality either with a randomized method or if given $\lg\lg$(relative frequency of the mode) with a constant additive error.

1 Introduction

The memory in modern computers consists of multiple levels. Traditionally, algorithms were analyzed in a flat random-access memory model (RAM) in which access times are uniform. However, the ever growing difference between access times of different levels of a memory hierarchy makes the RAM model ineffective. Hierarchical memory models have been introduced to tackle this problem. These models usually suffer from the complexity of having too many parameters which result in algorithms that are often too complicated and hardware configuration dependant.

The *cache-aware (DAM) model* [1] is the simplest of hierarchical memory models, taking into account only two memory levels. On the first level, there is a cache memory of size M which is divided into blocks of size B; on the second level there is an arbitrarily large memory with the same block size. A word must be present in the cache to be accessed. If it is not in the cache, we say a *cache miss/fault* has occurred, and in this case the block containing the requested word must be brought in from the main memory. In case, all blocks in the cache are occupied, a block must be chosen for eviction and replacement. Thus, a *block replacement policy* is necessary for any algorithm in the model.

The *cache-oblivious model*, which was introduced by Frigo et al. [2], is a simple hierarchical memory model and differs from the DAM model in that it avoids any hardware configuration parametrization, and in particular it is

not aware of M, B. This makes cache-oblivious algorithms independent of any hardware configuration. The block replacement policy is assumed to be the off-line optimal one. However, using a more realistic replacement policy such as the least recently used policy (LRU) increases the number of cache misses by only a factor of two if the cache size is also doubled [3]. It is known that if an algorithm in this model performs optimally on this two-level hierarchy, it will perform optimally on any level of a multiple-level hierarchy.

Cache complexity of an algorithm is the number of cache misses the algorithm causes or equivalently the number of block transfers it incurs between these two levels. In this paper, our concern is with the order of magnitude of the cache complexities of algorithms and we ignore constant factors. Following a usual practice in studying sorting problems, we make the so called *tall cache assumption*, that is, we assume the cache has size $M = \Omega(B^2)$. *Work complexity* of an algorithm is the complexity of the number of operations performed and, since here we focus on comparison-based problems, the work complexity is the number of comparisons the algorithm performs.

We will study three problems regarding the searching and sorting in multisets, namely duplicate elimination, sorting and determining the mode. A multiset is a generalization of a set in which repetition is allowed, so we can have several elements with the same key value. Suppose we are given a multiset of size N with k distinct elements whose multiplicities are $N_1, \ldots, N_k$ and set $f = \max_i N_i$.The problem of reducing the original multiset to the set of *distinct* elements is called duplicate elimination. The problem of sorting the elements of the multiset and outputting the list of elements in the sorted order, is called multisorting. The problem of finding the most frequent element (the mode) in a multiset, is called determining the mode.

Sorting can be studied in two models: In the first model, elements are such that one of the two equal elements can be removed and then later on, it can be copied from the other one to be regenerated again . In this model, we can keep only one copy of an element and throw away duplicates as encountered. However, in the second model, elements cannot be deleted and regenerated as elements are viewed complex objects with some satellite data in addition to their key values. Obviously, an entire object cannot be deleted just because it has a key equal to the key of another object. By multisorting, then, we infer the second, more difficult problem.

Munro and Spira [4] studied these problems in the comparison model and showed tight lower bounds for them. Their results are summarized in Table 1. Later, Arge et al. [5] proved how the lower bounds in the comparison model can be turned into lower bounds for the cache-aware model. They used the method to obtain lower bounds for the problems of determining the mode and duplicate elimination. A lower bound on the multisorting problem can be obtained similarly. A lower bound in the cache-aware model is certainly a lower bound in cache oblivious model as well. The lower bounds are summarized in Table 1. In the following sections, we give optimal cache-oblivious algorithms that all match these lower bounds. The optimal algorithm for determining the mode is random-

ized. The deterministic algorithm differs from the cache-complexity lower bound of a sublogarithmic factor or requires "a little hint".

	Comparisons	I/Os
Multisorting	$N \log N - \sum_{i=1}^{k} N_i \log N_i$	$\max\left\{ \frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B} - \sum_{i=1}^{k} \frac{N_i}{B} \log_{\frac{M}{B}} N_i,\ \frac{N}{B} \right\}$
Duplicate Elimination	$N \log N - \sum_{i=1}^{k} N_i \log N_i$	$\max\left\{ \frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B} - \sum_{i=1}^{k} \frac{N_i}{B} \log_{\frac{M}{B}} N_i,\ \frac{N}{B} \right\}$
Determining the Mode	$N \log \frac{N}{f}$	$\max\left\{ \frac{N}{B} \log_{\frac{M}{B}} \frac{N}{f},\ \frac{N}{B} \right\}$

Table 1. The lower bounds in the comparison model and the cache-oblivious model

2 Duplicate Elimination

Our duplicate removal technique is an adaptation of the lazy funnelsort [6] which sorts a set of (distinct) values. Hence, we first give a quick overview of the method in Section 2.1 and then present our modifications in Section 2.2.

2.1 Funnelsort

To sort a set of (distinct) values, Frigo et al. [2] proposed funnelsort which can be described as a cache-oblivious version of the mergesort. In funnelsort, the set of N input values are first divided into $N^{1/3}$ subsequences each of size $N^{2/3}$. Each subsequence is sorted recursively and then the sorted subsequences are merged by a data structure which is referred to as a $N^{1/3}$-*funnel*.

A k-funnel is indeed a k-way merger that takes k sorted subsequences, merges them, and outputs the sorted sequence. A k-funnel has an output buffer of size k^3 and k input buffers. As Figure 1 illustrates, a k-funnel is built recursively out of $\sqrt{k}$ $\sqrt{k}$-funnels (on the left) and one $\sqrt{k}$-funnel (on the right). There are $\sqrt{k}$ intermediate buffers of size $2k^{3/2}$ each. They are FIFO queues that form the output buffers of the left funnels and the input buffers of the right funnel. It can be proved by induction that a k-funnel occupies $O\left(k^2\right)$ space.

A k-funnel works recursively as follows. The k-funnel must output k^3 elements at any invocation. To do this, the right funnel is invoked many times each outputting $k^{3/2}$ values. Therefore the right funnel is eventually executed $k^{3/2}$ times to get k^3 elements. Before each invocation though, the k-buffer checks all its input buffers to see if they are more than half full. If any buffer is less than half full, the associated left funnel is invoked to fill-up its output buffer.

Later, Brodal and Fagerberg [6] simplified funnels by introducing the notion of *lazy funnelsort*. Their modification consists of relaxing the requirement that all input buffers must be checked before each invocation of a funnel. Rather, a buffer is filled up on demand, when it runs empty. This modification simplifies

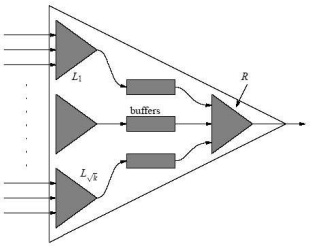

Fig. 1. ([2]) Recursive structure of a k-funnel.

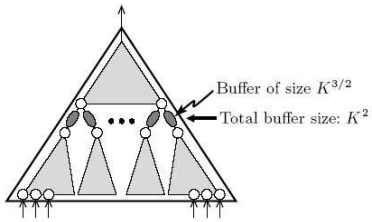

Fig. 2. ([7]) Structure of a lazy k-funnel.

the description of a funnel and as we will see in next sections, it also makes the analysis easier.

In lazy funnel sort, a k-funnel can be thought of as a complete binary tree with k leaves in which each node is a binary merger and edges are buffers. A tree of size S is laid out recursively in memory according to the van Emde Boas layout ([8]): The tree is cut along the edges at half height. The top tree of size $\sqrt{S}$ is recursively laid out first and it is followed by $\sqrt{S}$ bottom trees in order that are also laid out recursively. The sizes of the buffers can be computed by the recursive structure of a funnel. As it is illustrated in Figure 2, in a k-funnel the middle edges (at half height) have size $k^{3/2}$. The sizes of the buffers in the top and bottom trees (which are all $\sqrt{k}$-funnels) are recursively smaller. The buffers are stored along with the trees in the recursive layout of a funnel. For the sake of consistency, we suppose there is an imaginary buffer at the top of the root node to which the root merger outputs. The size of this buffer is k^3 (this buffer is not actually stored with the funnel). The objective is to fill up this buffer. At any node, we perform merging until either of the two input buffers run empty. In such a case, the merger suspends and control is given to the associated child to fill up the buffer recursively (the child is the root of its subtree).

The cache complexity analysis of a lazy k-funnel follows an amortized argument. Consider the largest value s such that an s-funnel along with one block from each of its input buffers fits in the cache memory. It is easy to see that under the tall cache assumption $s = \Omega(\sqrt[4]{M})$ and $s = O(\sqrt[2]{M})$. The whole k-funnel can be considered as a number of s-funnels that are connected by some large buffers among them. The sizes of these buffers are at least s^3. We denote these buffers as large buffers (see Figure 3).

An s-funnel fits entirely in cache memory and once completely loaded it does not cause any extra cache misses during its working. An s-funnel has size s^2 and thus it takes $O\left(s^2/B + s\right)$ cache faults to load it in memory. Though it outputs s^3 elements, thus the amortized cost per element that enters in the funnel is $O\left(1/B\right)$. However, in the event that its input buffers run empty, an s-funnel will have to stop and give control to its lower subfunnels. In that case, the funnel might get evicted from the memory and when its input buffers are filled up again,

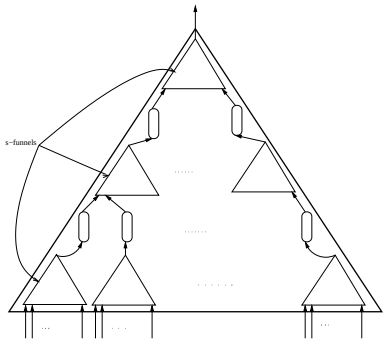

Fig. 3. The whole funnel can be considered as consisting of a number of s-funnels connected by some buffers.

it must be reloaded into memory. We have to account for these cache faults as well. Every time a buffer runs empty, it is filled up with at least s^3 elements. The cost of reloading the s-funnel can be charged to these s^3 elements. In the very last invocation a funnel might not be able to output enough number of elements, but we can simply charge the cost to the previous invocation (there exists at least one). Each element is thus charged $O(1/B)$ in each of the large buffers. Since there are $\Theta(\log_M N)$ such buffers, the total charge is $O\left(\frac{N}{B}\log_M N\right)$ which is optimal. The work complexity can be calculated similarly to be $O(N \log N)$.

2.2 Duplicate Removal by Funnels

As it was mentioned previously, our duplicate elimination is an adaptation of the lazy funnelsort. We introduce a constraint on a binary merger: When the two elements on top of the two input buffers of a binary merger are equal, one of them is appended to a memory zone devoted to contain the duplicates. In the end we obtain the set of distinct elements with all the duplicate elements laying in a separate zone in no particular order.

We now show that the work complexity of the above mentioned algorithm for duplicate elimination is optimal, i.e. $O(N \log N - \sum_i N_i \log N_i)$. We would spend $N \log N$ comparisons on finding the total ordering of the multiset, if elements were all pairwise distinct. Therefore, we need only to show that by removal of duplicates, we save $\sum_i N_i \log N_i$ number of comparisons. Consider the set E of duplicates of an element i_j ($|E| = N_j$). At any comparison of two elements in E, one of them is removed immediately, thus there can be at most N_j comparisons among elements of E. This implies a saving of $N_j \log N_j - N_j$ in the number of comparisons. Hence, in total we have a saving of $\sum_i N_i \log N_i - \sum_i N_i$. So the total number of comparisons is $O(N \log N) - \sum_i N_i \log N_i + \sum_i N_i = O(N \log N - \sum_i N_i \log N_i)$.

Now we show that the cache complexity of the algorithm also matches the lower bound shown in Section 1. The lower bound for the cache complexity of

duplicate elimination, by tall cache assumption, is $\Omega\left(\frac{N\log N-\sum_i N_i\log N_i}{B\log M}\right)$. The analysis is essentially the same as that of the cache complexity of funnelsort in Section 2.1, with the difficulty now that it is no longer true that it does exist at least one invocation of an s-funnel, with $s=\Omega(M^{1/4})$ and $s=O(M^{1/2})$, having in input at least s^3 elements.

As depicted in Fig. 2, a k-merger can be seen as a complete binary tree with k leaves, where every internal node is a binary merger and every edge is a buffer. The whole recursive algorithm can be then abstracted as a complete binary tree T with buffers on the edges. We know that every recursive call corresponds to a t-funnel, for some t, having t input buffers of size t^2 each (possibly containing less than t^2 elements because of duplicate removal) and an output buffer of size t^3. Given this view, the topmost $(1/3)\log N$ levels of T correspond to the $N^{1/3}$-funnel which is the last one executed in the algorithm. The leaves of this funnel have input buffers of size $N^{2/3}$, which are filled up by the $N^{2/9}$-funnels corresponding to the next $(2/9)\log N$ levels of T. These funnels are executed before the top one. The other levels of T are defined recursively.

To evaluate the cost of our algorithm's recursion, we observe that when the funnels constituting T are fully confined within main memory no extra I/O occurs. These *small* funnels lie in the bottommost $l=\Theta(\log M)$ levels of T, and they have size between $M^{2/3}$ and M. The *larger* funnels above the small ones in T thus contain $O(N/M^{2/3})$ overall binary mergers. Since we know that an s-funnel takes $O(s+s^2/B)$ extra I/Os to perform its duplicate elimination task, every binary merger in this funnel pays $O(1+s/B)$ extra I/Os. Summing over all the binary mergers of the larger funnels we have that the cache complexity of loading and using these s-funnels is $O(N/M^{2/3})O(1+s/B)=O(N/B)$. This shows that the cache complexity of an element entering in a s-funnel is $O\left(1/B\right)$.

By considering the path an element takes from a leaf of T to its root, we have that the element enters a number of s-funnels which is proportional to the number of comparisons it participates in divided by $\Omega\left(\log M\right)$. Since the overall number of comparisons the algorithm performs is $N\log N-\sum_i N_i\log N_i$ (see above), the number of element entrances into s-funnels is $O((1/\log M)(N\log N-\sum_i N_i\log N_i))$. But, as we argued, every element entrance in an s-funnel costs $O(1/B)$ and hence the cache complexity of the algorithm is $O\left(\frac{N\log N-\sum_i N_i\log N_i}{B\log M}\right)$.

Theorem 1. *The cache complexity of the duplicate removal algorithm matches the lower bound and is* $O\left(\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B}-\sum_{i=1}^{k}\frac{N_i}{B}\log_{\frac{M}{B}}N_i\right)$.

3 Multisorting

In this section, we will show how to sort a multiset within a constant factor of the optimal number of I/Os. We will match the lower bound presented in Section 1 which is the same as the lower bound for duplicate elimination.

The algorithm consists of two phases. The first phase is duplicate elimination, the second phase is based on a "reverse" application of a funnel now oriented

to *elements distribution* rather than to elements merging. We run the duplicate elimination algorithm to discover all the distinct elements in the multiset and to count their multiplicities as follows. We associate a counter to each element which is initialized to one in the beginning. When two duplicates are compared, one of them is appended to a memory zone devoted to contain the duplicates and we add its counter value to the counter of the element remained. Knowing the sorted order of the distinct elements and their multiplicities, we can easily figure out the boundaries of duplicates in the sorted multiset: The sorted multiset will be composed of subsequences $A_1A_2 \dots A_m$ where A_r consists entirely of duplicates of element i_r. By a scan over the sorted distinct elements, we find out the beginning and ending positions of each duplicate sequence A_i in the sorted multiset.

For the second phase, we need the concept of *k-splitter*. A k-splitter is a binary tree with k leaves. Each internal node v of a splitter is composed by an input buffer, a *pivot* $p(v)$ and a pointer to a contiguous memory portion $Z(v)$ *outside* the k-splitter where the duplicates of $p(v)$ will be moved as soon as they meet with the pivot. The root of the splitter, and leaves are, respectively, the input and the output buffers of the whole k-splitter. The buffers associated with the root and the leaves have sizes $O(k^3)$ and $O(k^2)$, respectively, and are considered external entities, just like the $Z(v)$, and they will not be accounted as space occupied by the whole k-splitter. As for the internal nodes, the choice of the size of their buffers, and the layout in memory of a k-splitter, is done as in the case of a k-merger and therefore the space occupied is $O(k^2)$ (only the buffers and pivots associated with internal nodes are accounted). When an internal node v is invoked, the elements of its buffer are partitioned and sent to its two children according to the results of comparisons with $p(v)$. The duplicates of the pivot are immediately moved into $Z(v)$. The process involving v stops when its buffer is empty, or if the buffer of at least one of its children is full: in the first case the parent of v is invoked, in the second case the child (children) corresponding to the full buffer(s) is (are) invoked. The root is marked as "done" when there are no more elements in its buffer. Any node other than the root is marked as "done" when there are no more elements in its buffer and its parent has been marked as done.

We are ready now to describe the second phase of our multisorting algorithm. Recall that the first phase produced two sequences, one containing the sorted set of distinct elements and the other one with the remaining duplicates in no particular order. Let us denote respectively with A and B these sequences. Moreover, for each element e we have counted its multiplicity $n(e)$. The second phase consists of three main steps.

1. With a simple scan of A a sequence $C = A_1A_2 \dots A_m$ is produced. For any i, A_i has $n(A[i])$ positions, its first position is occupied by $A[i]$ and any other position contains a pointer to the first location of subsequence A_i. Let $D = E_1E_2 \dots E_m$ be a contiguous memory portion divided into subarrays E_i of size $n(A[i])$. In the end, D will contain the sorted multiset. Finally, we let

$F = G_1 G_2 \dots G_m$ be a contiguous memory portion divided into subarrays G_i of size $n(A[i])$.

2. Using C we construct a splitter S of height at most $(1/3)\log N$ (not necessarily a complete splitter) and we invoke it. Let A_j be the subsequence of C that includes the $N/2$-th location (we need $O(1)$ I/Os to find j). If $j > 1$ or $j < m$ then we have the element $A_j[1]$ as the pivot of the root of the splitter. This step is applied recursively to the sequences $C' = A_1 \dots A_{j-1}$ and $C'' = A_{j+1} \dots A_m$ (at least one of them exists). That recursive process stops when the maximal tree S of pivots of height $(1/3)\log N$ is built (not necessarily a complete binary tree) or when there are no more sequences A_j. Note that each internal node v of S is associated to a subsequence A_j (the one from which $p(v)$ has been taken). Let t be the number of internal nodes of S, let $j_1, j_2 \dots j_t$ be the indices (in increasing order) of the sub-sequences A_j of C chosen in the process above, and let v_{j_r} be the internal node of S that has $A_{j_r}[1]$ as pivot. The memory zone $Z(v_{j_r})$ devoted to contain the duplicates of v_{j_r}, is E_{j_r}. The input buffer of the root of S is B (the one that contains the duplicates after the first phase). The buffer of the leaf u that is the right (resp. left) child of a node v_{j_r} of S is $B_u = G_{j_r+1} G_{j_r+2} \dots G_{j_{r+1}-1}$ (resp. $B_u = G_{j_{r-1}+1} G_{j_{r-1}+2} \dots G_{j_r-1}$). Finally, the sizes of the buffers of all the internal nodes and their layout in memory are chosen as we already discussed above when we gave the definition of k-splitter. Now, the splitter is ready and can be invoked.

3. Recursive steps. For any output buffer of the splitter S we apply the Step 2 recursively. Let u be a leaf of S and let us suppose that is the right child of a node v_{j_r} of S (the left-child case is analogous). In the recursive call for u we have that $B_u = G_{j_r+1} G_{j_r+2} \dots G_{j_{r+1}-1}$ plays the role of B, $E_{j_r+1} E_{j_r+2} \dots E_{j_{r+1}-1}$ plays the role of D, and $A_{j_r+1} A_{j_r+2} \dots A_{j_{r+1}-1}$ plays the role of C. After the recursive steps, D contains the sorted multiset.

The analysis is similar to the one for the duplicate elimination problem, and thus we get the same optimal I/O-bound. Indeed, the first step is a simple scan and requires $O(N/B)$ I/Os. In the second step, the splitter S can be divided in s-splitters pretty analogously to the case of s-mergers. All the "spurious" I/O's— like the $O(1)$ accesses for the construction of any node in the splitter or the $O(s + s^2/B)$ accesses needed to load a s-splitter— give again $O(N/B)$ I/Os.

Theorem 2. *The cache complexity of the multisorting algorithm matches the lower bound and is* $O\left(\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B} - \sum_{i=1}^{k}\frac{N_i}{B}\log_{\frac{M}{B}} N_i\right)$.

4 Determining the mode

In this section, we study the problem of determining the most occurring element (mode) in a multiset. We will take two approaches: Deterministic and Randomized. The upper bound we achieve in the randomized approach matches the lower bound for finding the mode as was mentioned in Section 1, essentially

because we can use samples to get a "good enough" estimate of the *relative frequency* of the mode (i.e. f/N). However, the deterministic approach can be in the worst case an additive term of $O\left(\frac{N}{B}\log\log M\right)$ away from the lower bound. The deterministic and randomized approaches are presented in the following two sections.

4.1 Deterministic Approach

The key idea is to use as a basic block a cache-efficient algorithm for finding "frequent" elements that occur more often than a certain threshold in the multiset. We then repeatedly run the algorithm for a *spectrum of thresholds* to hunt the most frequent element. Let us first precisely define what we mean by a "frequent" element.

Definition 1. *We call an element* C-frequent *if and only if it occurs more than* $\frac{N}{C}$ *times in a multiset of size* N *(i.e. if its relative frequency is at least* $1/C$*).*

The algorithm works in two phases. In the first phase, we try to find a set of at most C candidates that contains all the C-frequent elements. There may also be some other arbitrarily infrequent elements in our list of candidates. In the second phase, we check the C candidates to determine their exact frequencies. Note that, by definition, the number of C-frequent elements cannot exceed C.

Phase 1. The key idea in this phase is essentially what Misra [9] used. We find and remove a set of t ($t \geq C$) distinct elements from the multiset. The resulting multiset has the property that those elements that were C-frequent in the original multiset are still C-frequent in the reduced multiset. Thus, we keep removing sets of at least C distinct elements from the multiset, one at a time, until the multiset has no longer more than C distinct elements. The C-frequent elements in the original multiset must be also present in the final multiset.

We scan the multiset in groups of C elements (there are N/C such groups) by maintaining an array of C *"candidates"* which is initially empty and eventually will hold as many as C distinct values. Each element in this array also has a counter associated with it which shows the number of occurrences of the element seen so far. As soon as the number of elements in this array goes over C, we downsize the array by removing C distinct elements. More precisely, for each group G of C elements from the multiset, we first sort the elements in G and also sort the elements in the candidates array. This can be done by using any method of cache-oblivious sorting by pretending that the elements are all distinct. Then, we merge the two sorted arrays into another array T ensuring that we keep only the distinct elements. T may contain up to $2C$ elements. At this time we remove groups of at least C distinct elements to downsize T to less than C elements. This is done by sorting T according to the value of the counters (not the value of elements), and by finding the $C+1$ largest counter m_{C+1}. All elements with a counter value less than m_{C+1} are thrown away, and the counter of the other elements is decreased by m_{C+1}. One can easily see that this is equivalent to

repeatedly throwing away groups of at least C distinct elements from T one at a time. The candidates array is then set to T, and the process continues on the next group of C elements from the multiset. At the end, the candidates array contains all possible C-frequent elements in the multiset; however, as mentioned, it may also contain some other arbitrary elements.

Phase 2. This phase is similar to the first phase, except that the candidates array remains intact throughout this phase. We first zero out all the counters and sort the array using any method of cache-oblivious sorting for sets. We then consider N/C groups of C elements from the multiset one at a time; we first sort the C elements for each group, and by doing a scan of the lists as in the merge sort, we can count how many times each of the candidates occur in the group. We accumulate these counts so that after considering the final group, we know the multiplicities of the candidates in the whole multiset. We finally keep all elements whose counters are more than N/C and discard the rest.

Cache complexity of both phases are the same and one can see the major task is N/C executions of sorting C elements. Therefore, the cache complexity is $O\left(\frac{N}{B}\max\left\{1, \log_{\frac{M}{B}} C\right\}\right)$. Hence, we have proved the following lemma:

Lemma 1. *In a multiset of size N, C-frequent elements and their actual multiplicities can be determined with cache complexity $O\left(\frac{N}{B}\max\{1, \log_{\frac{M}{B}} C\}\right)$.*

Now we show how the frequent finding algorithm can be used in our hunt for the mode. We repeatedly apply Lemma 1 for a series of increasing values of C to determine whether there is any C-frequent element in the multiset. The first time some C-frequent elements are found, we halt the algorithm and declare the most frequent among them as the mode.

The algorithm in Lemma 1 is executed in rounds as C goes doubly exponentially for the following values of C in order: $C = 2^{2^1}, 2^{2^2}, \ldots, 2^{2^i}, \ldots, 2^{2^{\lceil \lg \lg n \rceil}}$. At the end of each round, we either end up empty-handed or we find some C-frequent elements. In the former case, the algorithm continues with the next value of C. In the latter case, we declare the most frequent of the C-frequent elements to be the mode, and the algorithm halts. Note that the algorithm of Lemma 1 also produces the actual multiplicities of the C-frequent elements, thus finding the most frequent element among the C-frequent ones requires only a pass of the at most C elements to select the element with the maximum multiplicity.

The cache complexity of the algorithm can be analyzed as follows. Let us denote by f the frequency of the mode. The cache complexity of the algorithm is the sum of cache complexity of the algorithm in Lemma 1 over different values of C up to $2^{2^{\lg \lg \frac{N}{f}+1}}$, where we find the mode. Hence, the cache complexity is

$$\sum_{j=1}^{\lg \lg \frac{N}{f}+1} O\left(\frac{N}{B}\max\left\{1, \log_{\frac{M}{B}} 2^{2^j}\right\}\right) = O\left(\max\left\{\frac{N}{B}\log\log\frac{N}{f}, \frac{N}{B}\log_{\frac{M}{B}}\frac{N}{f}\right\}\right).$$

It is clear to see that the following cache oblivious upper bound can be in the worst case an additive term of $O\left(\frac{N}{B}\log\log M\right)$ larger than the lower bound:

Theorem 3. *The cache complexity of the deterministic algorithm for determining the mode is* $O\left(\max\left\{\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{f}, \frac{N}{B}\log\log\frac{N}{f}\right\}\right)$. □

It is worthy to note that the slightest hint on the size of memory or the relative frequency of the mode would result in an optimal algorithm. Given the value of M, we can tailor the program so it skips from values of C smaller than M and starts from $C = M$. Thus, as a by-product, we have an optimal *cache-aware algorithm* which is simpler than the existing one in Arge et al. [5]. Also, knowing the value of $\lg\lg\frac{N}{f}$ with a constant additive error helps us to jump start from the right value for C. Furthermore, given an element, we can confirm with an optimal cache complexity whether it is indeed the mode. Let us define *having a hint* on a value v as knowing the value of $\lg\lg v$ within a constant additive error:

Theorem 4. *Given a hint on the value of memory size or relative frequency of the mode (i.e.* M *or* $\frac{N}{f}$*), the cache complexity of the deterministic algorithm for determining the mode matches the lower bound and is* $O\left(\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{f}\right)$.

4.2 Randomized Approach

We still use the C-frequent finder algorithm, but instead of starting from small values of C and squaring it at each step, we will estimate a good value for C using randomized sampling techniques. The sample must be large enough to produce a good estimate, with high confidence. It must also be small enough so that working with the sample does not dominate our cost. We have a high degree of latitude in choosing the sample size. Something around $\sqrt{N}$ is a reasonable choice. Making it $\sqrt{N}\ln N$ simplifies some of the calculations in the proof. The proof is also simplified if we sample with replacement (i.e. the same element can be chosen more than once.)

We can afford to sort these sampled elements as sorting $\sqrt{N}\ln N$ elements takes $O\left(\frac{N}{B}\right)$ cache misses. After sorting the sample, we scan and find the mode in the sample with frequency p. The estimate of the frequency of the mode in the multiset is $f' = \frac{p\sqrt{N}}{\ln N}$. Consequently we start by finding C-frequent elements for $C = \frac{N}{f'}$. If there is no C-frequent element, we square C and re-run the algorithm for the new value of C and so on.

Let us now sketch the analysis the cache complexity of the randomized algorithm. Clearly, if our estimate for the mode is precise, then the cache complexity of the algorithm matches the lower bound. However undershoot or overshoot are possible: We may underestimate the value of f (i.e. $f' < f$), or we may overestimate the value of f (i.e. $f' > f$). In the former case, we find the mode on the first run of the frequent element finder algorithm, but as the value of C is greater than what it should be, the cache complexity of the algorithm can be potentially larger. In the latter case, multiple runs of the frequent finder algorithm is likely; Therefore, potentially we can have the problem of too many runs as in the deterministic approach. Nevertheless, one can show that the probability of

our estimate for the mode being too far from the real value is small enough so that the extra work does not effect the expected asymptotic cache complexity. Hence, the expected cache complexity will match the lower bound.

Theorem 5. *The expected cache complexity of the randomized algorithm for determining the mode matches the lower bound and is* $O\left(\max\left\{\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{fB}, \frac{N}{B}\right\}\right)$.

5 Conclusion

We studied three problems related to multisets in the cache-oblivious model: duplicate removal, multi-sorting, and determining the mode. We presented the known lower bounds for the cache complexity of each of these problems. Determining the mode has the lower bound of $\Omega\left(\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{fB}\right)$ where f is the multiplicity of the most frequent element and M is the size of the cache and B is size of a block in cache. The lower bound for the cache complexity of duplicate removal and multi-sorting is $\Omega\left(\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B} - \sum_{i=1}^{k}\frac{N_i}{B}\log_{\frac{M}{B}}\frac{N_i}{B}\right)$.

The cache complexities of our algorithms match the lower bounds asymptotically. Only exception is the problem of determining the mode where our deterministic algorithm can be an additive term of $O\left(\frac{N}{B}\log\log M\right)$ away from the lower bound. However, the randomized algorithm matches the lower bound.

References

1. Aggarwal, A., Vitter, J.S.: The I/O complexity of sorting and related problems. In: Proceedings of the 14th International Colloquium on Automata, Languages, and Programming. Volume 267 of LNCS., Springer-Verlag (1987) 467–478
2. Frigo, M., Leiserson, C.E., Prokop, H., Ramachandran, S.: Cache-oblivious algorithms. In: 40th Annual Symposium on Foundations of Computer Science, IEEE Computer Society Press (1999) 285–297
3. Sleator, D.D., Tarjan, R.E.: Amortized efficiency of list update and paging rules. Commun. ACM **28(2)** (1985) 202–208
4. Munro, I., Spira, P.: Sorting and searching in multisets. SIAM Journal on Computing **5** (1976) 1–8
5. Arge, L., Knudsen, M., Larsen, K.: A general lower bound on the i/o-complexity of comparison-based algorithms. In: In Proceedings of Workshop on Algorithms and Data Structures, Springer-Verlag (1993)
6. Brodal, Fagerberg: Cache oblivious distribution sweeping. In: ICALP. (2002)
7. Demaine, E.D.: Cache-oblivious algorithms and data structures. In: Lecture Notes from the EEF Summer School on Massive Data Sets. Lecture Notes in Computer Science, BRICS, University of Aarhus, Denmark (2002)
8. Bender, M.A., Demaine, E.D., Farach-Colton, M.: Cache-oblivious B-trees. In IEEE, ed.: Annual Symposium on Foundations of Computer Science 2000, IEEE Computer Society Press (2000) 399–409
9. Misra, J., Gries, D.: Finding repeated elements. Science of Computer Programming **2** (1982) 143–152