Cache-Aware and Cache-Oblivious Adaptive Sorting

Gerth Stølting Brodal[1,⋆], Rolf Fagerberg[2,⋆⋆], and Gabriel Moruz[1]

[1] BRICS⋆⋆⋆, Department of Computer Science, University of Aarhus, IT Parken, Åbogade 34, DK-8200 Århus N, Denmark. E-mail: {gerth,gabi}@daimi.au.dk
[2] Department of Mathematics and Computer Science, University of Southern Denmark, Campusvej 55, DK-5230 Odense M, Denmark. E-mail: rolf@imada.sdu.dk

Abstract. Two new adaptive sorting algorithms are introduced which perform an optimal number of comparisons with respect to the number of inversions in the input. The first algorithm is based on a new linear time reduction to (non-adaptive) sorting. The second algorithm is based on a new division protocol for the GenericSort algorithm by Estivill-Castro and Wood. From both algorithms we derive I/O-optimal cache-aware and cache-oblivious adaptive sorting algorithms. These are the first I/O-optimal adaptive sorting algorithms.

1 Introduction

1.1 Adaptive sorting

A well known fact concerning sorting is that optimal sorting algorithms perform $\Theta(n \log n)$ comparisons [9, Section 9.1]. However, in practice there are many cases where the input sequences are already nearly sorted, i.e. have low disorder according to some measure [16, 19]. In such cases one can hope for a sorting algorithm to be faster.

In order to quantify the disorder of input sequences, several *measures of presortedness* have been proposed, e.g. see [11, 16, 18]. One of the most commonly considered measures is *Inv*, the number of inversions in the input, defined by $Inv(X) = |\{(i,j) \mid i < j \ \wedge x_i > x_j\}|$ for a sequence $X = (x_1, \ldots, x_N)$. Other examples of measures include: *Runs*, the number of boundaries between ascending subsequences; *Max*, the largest difference between the ranks of an element in the input and the sorted sequence; *Dis*, the largest distance determined by an inversion. A sorting algorithm is denoted *adaptive* if the time complexity is a function dependent on the size as well as the presortedness of the input sequence [19]. For an overview concerning adaptive sorting, see e.g. the survey by Estivill-Castro and Wood [13].

⋆ Supported by the Carlsberg Foundation (contract number ANS-0257/20) and the Danish Natural Science Foundation (SNF).
⋆⋆ Partially supported by the Danish Natural Science Foundation (SNF).
⋆⋆⋆ Basic Research in Computer Science, www.brics.dk, funded by the Danish National Research Foundation.

Manilla [18] introduced the concept of optimality of an adaptive sorting algorithm in the comparison model. An adaptive sorting algorithm S is optimal with respect to some measure of presortedness $\mathcal{D}$, if for some constant $c > 0$ and for all inputs X, the time complexity $T_A(X)$ satisfies

$$T_A(X) \leq c \cdot \max(N, \log |\text{below}(X, \mathcal{D})|) \ ,$$

where $\text{below}(X, \mathcal{D})$ is the number of permutations of the input sequence Y for which $\mathcal{D}(Y) \leq \mathcal{D}(X)$ and $\log x$ denotes $\log_2 x$. By the usual information theoretic lower bound, this is asymptotically the best possible. In particular, an adaptive sorting algorithm that is optimal with respect to the measure *Inv* performs $\Theta(N(1 + \log(1 + Inv/N)))$ comparisons [15].

1.2 The I/O model and the cache-oblivious model

Traditionally, the RAM model has been used in the design and analysis of algorithms. It consists of a CPU and an infinite memory, where all memory accesses are assumed to take equal time. However, this model is not always adequate in practice, due to the memory hierarchy found on modern computers. Modern computers have several memory levels, each level having smaller size and access time than the next one. Typically, a desktop computer contains CPU registers, L1, L2, and L3 caches, main memory and hard-disk. The access time increases from one cycle for registers and level 1 cache to around 10, 100 and 10,000,000 cycles for level 2 cache, main memory and disk, respectively. Therefore, the I/Os of the disk often become a bottleneck with respect to the running time of a given algorithm, and the number of I/Os, not CPU cycles, should be minimized.

Several models have been proposed to capture the effect of memory hierarchies. The most successful of these is the *I/O model*, introduced by Aggarwal and Vitter [1]. It models a simple two-level memory hierarchy consisting of a fast memory of size M and a slow infinite memory. The data transfers between the slow and fast memory are performed in *blocks* of size B of consecutive data. The I/O complexity of an algorithm is the number of transfers it performs between the slow and the fast memories. A comprehensive list of I/O efficient algorithms for different problems have been proposed, e.g. see the surveys by Vitter [21] and Arge [2]. Among the fundamental results concerning the I/O model is that sorting a sequence of size N requires $\Theta(\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B})$ I/Os [1].

The I/O model assumes that the size M of the fast memory and the block size B are known, which does not always hold in practice. Moreover, as the modern computers have multiple memory levels with different sizes and block sizes, different parameters are required at the different memory levels. Frigo et al. [14] proposed the *cache-oblivious model*, which is similar to the I/O model, but assumes no knowledge about M and B. In short, a cache-oblivious algorithm is an algorithm described in the RAM model, but analyzed in the I/O model with an analysis valid for any values of M and B. The power of this model is that if a cache-oblivious algorithm performs well on a two-level memory hierarchy with arbitrary parameters, it performs well between all the consecutive levels of a multi-level memory hierarchy.

Many problems have been addressed in the cache-oblivious model (see the surveys by Arge et al. [3], Brodal [6], and Demaine [10]). Among these there are several optimal cache-oblivious sorting algorithms. Frigo et al. [14] gave two optimal cache-oblivious algorithms for sorting: *Funnelsort* and a variant of *Distributionsort*. Brodal and Fagerberg [7] introduced a simplified version of Funnelsort, *Lazy Funnelsort*. The I/O complexity of all these sorting algorithms is $O(\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B})$. All these algorithms require a *tall cache* assumption, i.e. $M = \Omega(B^{1+\varepsilon})$ for a constant $\varepsilon > 0$. In [8] it is shown that a tall cache-assumption is required for all optimal cache-oblivious sorting algorithms.

1.3 Results and outline of paper

In Section 2 we apply the lower bound technique from [4] to obtain lower bounds on the number of I/Os for comparison based sorting algorithms that are adaptive with respect to different measures of presortedness.

In Section 3 we present a linear time reduction from adaptive sorting to general (non-adaptive) sorting, directly implying comparison optimal and I/O-optimal cache-aware and cache-oblivious algorithms with respect to measure *Inv*.

In Section 4 we describe a cache-aware generic sorting algorithm, *cache-aware GenericSort* based on *GenericSort*, introduced in [12], and characterize its I/O adaptiveness. Section 5 introduces a cache-oblivious version of GenericSort.

In Section 6 we introduce a new greedy division protocol for GenericSort, interesting in its own right due to its simplicity. We prove that the resulting algorithm, *GreedySort*, is comparison optimal with respect to measure *Inv*. We show that using our division protocol we obtain both cache-aware and cache-oblivious algorithms that are optimal with respect to *Inv*.

In the remainder of this paper, sorted means sorted in increasing order.

2 I/O lower bounds

In this section we show lower bounds on the number of I/Os performed by comparison based sorting algorithms that are adaptive with respect to several measures of presortedness.

Theorem 1. *A comparison based sorting algorithm performs* $\Omega(\frac{N}{B}(1+\log_{\frac{M}{B}}(1+\frac{Inv}{N})))$ *I/Os for sorting input sequences of size* N *and* Inv *inversions, assuming* $M = \Omega(B^2)$.

Proof. Consider an adaptive sorting algorithm A and some input sequence X of size N. Let $T_A(X)$ and $I/O_A(X)$ denote the number of comparisons and the number of I/Os performed by a comparison based sorting algorithm A for sorting an input sequence X respectively.

Recall that below(X, Inv) denotes the set of all permutations Y for the input sequence with $Inv(Y) \leq Inv(X)$. Consider the decision tree of A (see e.g. [9, Section 9.1]) restricted to the inputs in below(X, Inv). The tree has at least

Measure of presortedness	I/Os	Comparisons [13]
Dis	$\Omega\left(\frac{N}{B}\left(1+\log_{\frac{M}{B}}(1+Dis)\right)\right)$	$\Omega(N(1+\log(1+Dis)))$
Exc	$\Omega\left(\frac{N}{B}\left(1+Exc\log_{\frac{M}{B}}(1+Exc)\right)\right)$	$\Omega(N+Exc\log(1+Exc))$
Enc	$\Omega\left(\frac{N}{B}\left(1+\log_{\frac{M}{B}}(1+Enc)\right)\right)$	$\Omega(N(1+\log(1+Enc)))$
Inv	$\Omega\left(\frac{N}{B}\left(1+\log_{\frac{M}{B}}\left(1+\frac{Inv}{N}\right)\right)\right)$	$\Omega\left(N\left(1+\log\left(1+\frac{Inv}{N}\right)\right)\right)$
Max	$\Omega\left(\frac{N}{B}\left(1+\log_{\frac{M}{B}}(1+Max)\right)\right)$	$\Omega(N(1+\log(1+Max)))$
Osc	$\Omega\left(\frac{N}{B}\left(1+\log_{\frac{M}{B}}\left(1+\frac{Osc}{N}\right)\right)\right)$	$\Omega\left(N\left(1+\log\left(1+\frac{Osc}{N}\right)\right)\right)$
Reg	$\Omega\left(\frac{N}{B}\left(1+\log_{\frac{M}{B}}(1+Reg)\right)\right)$	$\Omega(N(1+\log(1+Reg)))$
Rem	$\Omega\left(\frac{N}{B}\left(1+Rem\log_{\frac{M}{B}}(1+Rem)\right)\right)$	$\Omega(N+Rem\log(1+Rem))$
$Runs$	$\Omega\left(\frac{N}{B}\left(1+\log_{\frac{M}{B}}(1+Runs)\right)\right)$	$\Omega(N(1+\log(1+Runs)))$
SMS	$\Omega\left(\frac{N}{B}\left(1+\log_{\frac{M}{B}}(1+SMS)\right)\right)$	$\Omega(N(1+\log(1+SMS)))$
SUS	$\Omega\left(\frac{N}{B}\left(1+\log_{\frac{M}{B}}(1+SUS)\right)\right)$	$\Omega(N(1+\log(1+SUS)))$

Fig. 1. Lower bounds on the number of I/Os and the number of comparisons.

$|\text{below}(X, Inv)|$ leaves and therefore A performs at least $\log|\text{below}(X, \mathcal{D})|$ comparisons in the worst case. Therefore, for any sequence X, there is a sequence $Y \in \text{below}(X, Inv)$, such that $\log|\text{below}(X, Inv)| \leq T_A(Y)$.

Using the decision tree translation by Arge et al. [4, Theorem 1] we get:

$$\log(|\text{below}(X, Inv)|) \leq N\log B + \max_{Y\in\text{below}(X,Inv)} I/O_A(Y)\left(B\log\left(\frac{M}{B}\right)+3B\right).$$

Since $\log(|\text{below}(X, Inv)|) = \Omega(N(1+\log(1+\frac{Inv}{N})))$ [15], we obtain that $\max_{Y\in\text{below}(X,Inv)} I/O_A(Y) = \Omega(\frac{N}{B}(1+\log_{\frac{M}{B}}(1+\frac{Inv}{N})))$, given $M = \Omega(B^2)$. □

Using a similar technique we obtain lower bounds on the number of I/Os for other measures of presortedness, assuming that $M = \Omega(B^2)$. Figure 1 lists these lower bounds. For definitions of the different measures, refer to [13].

3 GroupSort

In this section we describe a reduction to derive Inv adaptive sorting algorithms from non-adaptive sorting algorithms. The reduction is cache-oblivious and requires $O(N)$ comparisons and $O(N/B)$ I/Os.

The basic idea is to distribute the input sequence into a sequence of buckets $S_1, \ldots, S_k$ each of size at most $32(Inv/N)^2$, where the elements in bucket S_i are all smaller than or equal to the elements in S_{i+1}. Each S_i is then sorted independently by a non-adaptive cache-oblivious sorting algorithm [7, 14]. During

```
procedure GroupSort(X)
Input: Sequence X = (x_1, ..., x_N)
Output: Sequence X sorted
begin
    S_1 = (x_1); F_1 = (); β_1 = 8; α_1 = N/4; j = 1; k = 1;
    for i = 2 to N
        if k = 1 or x_i ≥ min(S_k)
            append(S_k, x_i);
            if |S_k| > β_j
                (S_k, S_{k+1}) = split(S_k); k = k + 1;
        else
            append(F_j, x_i);
            if |F_j| > α_j
                β_{j+1} = β_j · 4; α_{j+1} = α_j/2; j = j + 1;
                while k > 1 and |S_k| < β_j/2
                    S_{k-1} = concat(S_{k-1}, S_k); k = k − 1;
    S = concat(sort(S_1), sort(S_2), ..., sort(S_k));
    F = concat(F_1, F_2, ..., F_j);
    GroupSort(F);
    X = merge(S, F);
end
```

Fig. 2. Linear time reduction to non-adaptive sorting.

the construction of the buckets $S_1, \ldots, S_k$ some elements might fail to get inserted into an S_i and are instead inserted into a *fail set* F. It will be guaranteed that at most half of the elements are inserted into F. The fail set F is sorted recursively and merged with the sequence of sorted buckets.

The S_i buckets are constructed by scanning the input left-to-right by inserting an element x into the rightmost bucket S_k if $k = 1$ or $x \geq \min(S_k)$ and otherwise inserting x in F. During the construction we generate increasing bucket capacities $\beta_j = 2 \cdot 4^j$, which will be used for $\alpha_j = N/(2 \cdot 2^j)$ insertions into F. If during construction $|S_k| > \beta_j$, the bucket S_k is split into two buckets S_k and S_{k+1} by computing its median using the cache-oblivious selection algorithm from [5] and distributing its elements relatively to the median. This ensures $|S_i| \leq \beta_j$ for $1 \leq i \leq k$. We maintain the invariant $|S_k| \geq \beta_j/2$ if there are at least two buckets by repeatedly concatenating the two last buckets after an increment of i. Since $\beta_{j-1} = \beta_j/4$, this ensures $\beta_j/2 \leq |S_k| \leq \frac{3}{4}\beta_j$ after this concatenation process. If only one bucket remains, then $|S_k| \leq \frac{3}{4}\beta_j$.

The pseudo-code of the reduction is given in Figure 2. We assume that $S_1, \ldots, S_k$ are stored consecutively in an array by storing the start index and the minimum element from each bucket on a separate stack, i.e. the concatenation of S_{k-1} and S_k can be done implicitly in $O(1)$ time. The fail set F is stored as a list of subsets $F_1, \ldots, F_j$, where F_i stores the elements inserted into F while the bucket size is β_i. Similarly $F_1, \ldots, F_j$ are stored consecutively in an array.

Theorem 2. GroupSort *is cache-oblivious and is comparison optimal and I/O-optimal with respect to Inv, assuming* $M = \Omega(B^2)$.

Proof. Consider the last bucket capacity β_j and fail set size α_j. Each element x inserted into the fail set F_j induces in the input sequence at least $\beta_j/2$ inversions, since $|S_k| \geq \beta_j/2$ when x is inserted into F_j and all elements in S_k appeared before x in the input and are larger than x.

For $i = \lceil \log \frac{Inv}{N} \rceil + 1$, we have $\alpha_i \cdot \frac{\beta_i}{2} = \frac{N}{2 \cdot 2^i} \cdot \frac{2 \cdot 4^i}{2} \geq Inv$, i.e. F_i is guaranteed to be able to store all failed elements. This immediately leads to $j \leq \lceil \log \frac{Inv}{N} \rceil + 1$, and $\beta_j = 2 \cdot 4^j \leq 32 \left(\frac{Inv}{N}\right)^2$. The fail set F has size at most $\sum_{i=1}^{j} \alpha_i = \sum_{i=1}^{j} N/(2 \cdot 2^i) \leq N/2$.

Taking into account that the total size of the fail sets is at most $N/2$, the number of comparisons performed by GroupSort is given by the following recurrence:

$$T(N) = T\left(\frac{N}{2}\right) + \sum_{i=1}^{k} T_{\mathrm{Sort}}(|S_i|) + O(N) \ ,$$

where the $O(N)$ term accounts for the bucket splittings and the final merge of S and F. The $O(N)$ term for splitting buckets follows from that when a bucket with β_j elements is split then at least $\beta_j/4$ elements in a bucket have been inserted since the most recent bucket splitting or increase in bucket capacity, and we can charge the splitting of the bucket to these recent $\beta_j/4$ elements.

Since $T_{\mathrm{Sort}}(N) = O(N \log N)$ and each $|S_i| \leq \beta_j = O((\frac{Inv}{N})^2)$ the number of comparisons performed by GroupSort is:

$$T(N) = T\left(\frac{N}{2}\right) + O\left(N\left(1 + \log\left(1 + \left(\frac{Inv}{N}\right)^2\right)\right)\right) \ .$$

Since F is a subsequence of the input, Inv for the recursive call is at most Inv for the input. As $\sum_{i=0}^{\infty} \frac{N}{2^i} \log \frac{Inv}{N/2^i} = N \log \frac{Inv}{N} \sum_{i=0}^{\infty} \frac{1}{2^i} + N \sum_{i=0}^{\infty} \frac{i}{2^i}$, it follows that GroupSort performs $T(N) = O\left(N\left(1 + \log\left(1 + \frac{Inv}{N}\right)\right)\right)$ comparisons, which is optimal.

The cache-oblivious selection algorithm from [5] performs $O(N/B)$ I/Os and the cache-oblivious sorting algorithms [7, 14] perform $O(\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B})$ I/Os for $M = \Omega(B^2)$. Since GroupSort otherwise does sequential access to the input and data structures, we get that GroupSort is cache-oblivious and the number of I/Os performed is given by the recurrence:

$$I/O(N) = I/O\left(\frac{N}{2}\right) + O\left(\frac{N}{B}\left(1 + \log_{\frac{M}{B}}\left(1 + \left(\frac{Inv}{N}\right)^2 \cdot \frac{1}{B}\right)\right)\right) \ .$$

It follows that GroupSort performs $O(\frac{N}{B}(1 + \log_{\frac{M}{B}}(1 + \frac{Inv}{N})))$ I/Os provided $M = \Omega(B^2)$, which by Theorem 1 is I/O-optimal. □

Pagh et al. [20] gave a related reduction for adaptive sorting on the RAM model. Their reduction assumes that a parameter q is provided such that the

number of inversions is at most qN. A valid q is found by selecting increasing values for q such that the running time doubles for each iteration. In the cache oblivious setting the doubling approach fails, since the first q value should depend on the unknown parameter M. We circumvent this limitation of the doubling technique by selecting the increasing β_j values internally in the reduction.

4 Cache-aware GenericSort

Estivill-Castro and Wood [12] introduced a generic sorting algorithm, *GenericSort*, as a framework for adaptive sorting algorithms. It is a generalization of Mergesort, and is described using a generic division protocol, i.e. an algorithm for splitting an input sequence into two or more subsequences. The algorithm works as follows: consider an input sequence X; if X is sorted then the algorithm returns; if X is "small", then X is sorted using some alternate non-adaptive sorting algorithm; otherwise, X is divided according to the division protocol and the resulting subsequences are recursively sorted and merged.

In this section we modify GenericSort to achieve a generic I/O-adaptive sorting algorithm. Consider an input sequence $X = (x_1, \ldots, x_N)$ and some division protocol DP such that DP splits the input in $s \geq 2$ subsequences of roughly equal sizes in a single scan, visiting each element of the input exactly once. To avoid testing whether X is sorted before applying the division protocol, we derive a new division protocol DP' by modifying DP to identify the longest sorted prefix of X: we scan the input sequence until we find some i such that $x_i < x_{i-1}$. Denote $S = (x_1, \ldots, x_{i-1})$ and $X' = (x_i, \ldots, x_N)$. We apply DP to X', recursively sort the resulting s subsequences, and finally merge them with S. The adaptive bounds for GenericSort proved in [12, Theorem 3.1] are not affected by these modifications, and we have the following theorem.

Theorem 3. *Let $\mathcal{D}$ be a measure of presortedness, d and s constants, $0 < d < 2$, and DP a division protocol that splits some input sequence of size N into s subsequences of size at most $\lceil \frac{N}{s} \rceil$ each using $O(N)$ comparisons.*

- *the modified GenericSort performs $O(N \log N)$ comparisons in the worst case;*
- *if for all sequences X, the division of a suffix of X into $X_1, \ldots, X_s$ by DP satisfies that $\sum_{j=1}^{s} \mathcal{D}(X_j) \leq d\lfloor \frac{s}{2} \rfloor \cdot \mathcal{D}(X)$, then the modified GenericSort performs $O\left(N\left(1 + \log(1 + \mathcal{D}(X))\right)\right)$ comparisons.*

We now describe a cache-aware version of the modified GenericSort provided that the division protocol DP works in a single scan of the input. Let T be the recursion tree of GenericSort using the new division protocol DP'. We obtain a new tree T' by contracting T top-down such that every node in T' corresponds to a subtree of height $O(\log_s(M/B))$ in T and each node in T' has a fanout of at most m, where $m = \Theta(M/B)$. There are $O(m)$ sorted prefixes for every node in T'. In cache-aware GenericSort, for each node of T' we scan its input sequence and distribute the elements accordingly to one of the $O(m)$ output sequences.

Each output sequence is a linked list of blocks of size $\Theta(B)$. If the size of the input sequence is at most M, then we sort it in internal memory, hence performing $O(N/B)$ I/Os. Theorem 4 gives a characterization of the adaptiveness of cache-aware GenericSort in the I/O model. It is an I/O version of Theorem 3.

Theorem 4. *Let $\mathcal{D}$ be a measure of presortedness, d and s constants, $0 < d < 2$ and $s \leq \frac{M}{2B}$, and DP a division protocol that splits some input sequence of size N into s subsequences of size at most $\lceil \frac{N}{s} \rceil$ each using $O(\frac{N}{B})$ I/Os. If DP performs the splitting in one scan visiting each element of the input exactly once, then:*

- *cache-aware GenericSort performs $O(\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B})$ I/Os in the worst case;*
- *if for all sequences X, the division of a suffix of X into $X_1, \ldots, X_s$ by DP satisfies that $\sum_{j=1}^{s} \mathcal{D}(X_j) \leq d\lfloor \frac{s}{2} \rfloor \cdot \mathcal{D}(X)$, then cache-aware GenericSort performs $O\left(\frac{N}{B}\left(1 + \log_{\frac{M}{B}}(1 + \mathcal{D}(X))\right)\right)$ I/Os.*

Proof. We analyze the I/Os performed at the nodes of T' separately for the nodes having input sizes less than or equal to M and greater than M.

At a node with input X and $|X| > M$, $O(m + |X|/B) = O(|X|/B)$ I/Os are performed to read the input and to write to the at most $m - 1$ sorted output prefixes and m sequences to be recursively sorted. If we charge $O(1/B)$ I/Os per element in the input this will pay for the I/Os required at the node.

At a node with input X and $|X| \leq M$, $O(1 + |X|/B)$ I/Os are performed. These I/Os can be charged to the parent node, since at the parent we will already charge $O(1 + |X|/B)$ I/Os to write the output X.

By Theorem 3 we have that the sum of the depths in T reached by the elements in the input X is bounded by $O(N(1 + \log(1 + \mathcal{D}(X))))$. Since each node in T' spans $\Theta(\log \frac{M}{B})$ levels from T, we get that cache-aware GenericSort performs $O(\frac{N}{B} + N(1+\log(1+\mathcal{D}(X)))/(B \log \frac{M}{B})) = O(\frac{N}{B}(1+\log_{\frac{M}{B}}(1+\mathcal{D}(X))))$ I/Os, where the N/B term counts for the I/Os at the root of T'. □

The power of cache-aware GenericSort lies in its generality, meaning that using different division protocols we obtain sorting algorithms that are I/O adaptive with respect to different measures of presortedness. For example, using the straight division protocol, we achieve I/O optimality with respect to *Runs*. Using the odd-even division protocol, we obtain an algorithm that is I/O optimal with respect to *Dis* and *Max*. Furthermore, the different division protocols can be combined as shown in [13] in order to achieve I/O optimality with respect to more measures of presortedness.

5 Cache-oblivious GenericSort

We give a cache-oblivious algorithm that achieves the same adaptive bounds as the cache-aware GenericSort introduced in Section 4. It works only for division protocols that split the input into two unsorted subsequences. It is based on a modification of the k-merger used in FunnelSort [7, 14].

A k-merger is a binary tree stored using the recursive van Emde Boas layout. The edges contain buffers of variable sizes and the nodes are binary mergers. The tree and the buffer sizes are recursively defined: consider an output sequence of size k^3 and h the height of the tree. We split the tree at level $\frac{h}{2}$ yielding $k^{\frac{1}{2}}+1$ subtrees, each of size $O(k^{\frac{1}{2}})$. The buffers at this level have sizes $k^{\frac{3}{2}}$. See [7] for further details.

Consider DP division protocol that scans the input a single time and DP' the modified DP as introduced in Section 4. Each node of the k-merger corresponds to a node in the recursion tree of GenericSort using DP' as the division protocol. Therefore, each node has a fanout of three and becomes a ternary merger. The resulting unsorted sequences are pushed in the buffers to the children, while the sorted prefix is stored as a list of memory chunks of size $O(N^{\frac{2}{3}})$ for an input buffer of size N.

Our algorithm uses a single $N^{\frac{1}{3}}$-merger. It fills the buffers in a top-down fashion and then merges the resulted sorted subsequences in a bottom-up manner. The $N^{\frac{1}{3}}$ output buffers at the leaves of the k-merger are sorted using a non-adaptive I/O-optimal cache oblivious sorting algorithm [7, 14].

Lemma 1. *The $N^{\frac{1}{3}}$-merger and the sorted subsequences use $O(N)$ space.*

Proof. Consider the $N^{\frac{1}{3}}$-merger and an input sequence of size N. The total size of the inner buffers is $O(N^{\frac{2}{3}})$ [14]. The memory chunks storing the sorted subsequences use $O(N)$ space because there are $N^{\frac{1}{3}}$ nodes in the merger and the size of a single memory chunk is $O(N^{\frac{2}{3}})$. Adding the input sequence, we conclude that the $N^{\frac{1}{3}}$-merger and the sorted subsequences take $O(N)$ space together. □

Lemma 2. *Cache-oblivious GenericSort and cache-aware GenericSort have the same comparison and I/O complexity, for division protocols that split the input into two subsequences.*

Proof. Consider $\ell = \frac{1}{3}\log N$ the height of the $N^{\frac{1}{3}}$-merger of the cache-oblivious GenericSort.

We first prove that cache-aware and cache-oblivious GenericSort have the same comparison complexity. For some element x_i let d_i be its depth in the recursion tree of the GenericSort using DP' as a division protocol. If $d_i \leq \ell$ then x_i reaches the same level in the recursion tree of cache-oblivious GenericSort, because the two algorithms have the same recursion trees at the top ℓ levels. If $d_i > \ell$ then the number of comparisons performed by cache-oblivious GenericSort for x_i is $O(\log N) = O(d_i)$ because $d_i > l = \Omega(\log N)$.

We analyze the number of I/Os used by cache-aware and cache-oblivious GenericSort. Consider an element x_i that reaches level d_i in the recursion tree of cache-aware GenericSort.

If $d_i < \ell$ then x_i is placed in a sorted prefix at a node in the $N^{\frac{1}{3}}$-merger. In this case, cache-oblivious GenericSort spends linear I/Os when the size of the input reaches $O(M)$ because the $N^{\frac{1}{3}}$-merger together with the sorted subsequences take linear space by Lemma 1. Taking into account that the height

of the $N^{\frac{1}{3}}$-merger is $O(\log(M/B))$ due to the tall cache assumption, it follows that $O(1 + d_i/(\log(M/B)))$ I/Os are performed by cache-oblivious GenericSort for getting x_i to its sorted subsequence.

If $d_i > \ell$ then x_i reaches an output buffer of the $N^{\frac{1}{3}}$-merger, where it is sorted using an optimal cache-oblivious sorting algorithm. In this case the number of I/Os performed for the sorting involving x_i is still $O(1/B + d_i/(B \log(M/B)))$, because both the $N^{\frac{1}{3}}$-merger and the optimal sorting algorithms require $O(1/B + d_i/(B \log(M/B)))$ I/Os for the sorting involving x_i, since $d_i = \Theta(\log N)$.

We obtain that the number of I/Os performed by cache-oblivious GenericSort is $O\left(\frac{N}{B} + \frac{\sum_{i=1}^{n} d_i}{B \log(M/B)}\right)$. Cache-aware GenericSort performs $O\left(\frac{N}{B} + \frac{\sum_{i=1}^{n} d_i}{B \log \frac{M}{B}}\right)$ I/Os too because the fanout of the nodes in the recursion tree is $O(\log \frac{M}{B})$. We conclude that cache-aware GenericSort and cache-oblivious GenericSort have the same I/O complexity. □

6 GreedySort

We introduce *GreedySort*, a sorting algorithm based on GenericSort using a new division protocol, *GreedySplit*. The protocol is inspired by a variant of the Kim-Cook division protocol, which was introduced and analyzed in [17]. Our division protocol achieves the same adaptive performance with respect to *Inv*, but is simpler and moreover facilitates cache-aware and cache-oblivious versions. It may be viewed as being of a greedy type, hence the name. We first describe GreedySort and its division protocol and then prove that it is optimal with respect to *Inv*. GreedySplit partitions the input sequence X into three subsequences S, Y, and Z, where S is sorted and Y and Z have balanced sizes, i.e. $|Z| \leq |Y| \leq |Z| + 1$. In one scan it builds an ascending subsequence S of the input in a greedy fashion and at the same time distributes the remaining elements in two subsequences, Y and Z, using an odd-even approach.

Lemma 3. *GreedySplit splits an input sequence X in the three subsequences S, Y and Z, where S is sorted and $Inv(X) \geq \frac{5}{4} \cdot (Inv(Y) + Inv(Z))$.*

Proof. Let $X = (x_1, \ldots x_N)$. By construction S is sorted. Consider an inversion in Y, $y_i > y_j$, $i < j$ and i_1 and j_1 the indices in X of y_i and y_j respectively. Due to the odd-even construction of Y and Z, there exists an $x_k \in Z$ such that in the original sequence X we have $i_1 < k < j_1$.

We prove that there is one inversion between x_k and at least one of x_{i_1} and x_{j_1}, for any $i_1 < k < j_1$. Indeed, if $x_{i_1} > x_k$, we get an inversion between x_{i_1} and x_k. If $x_{i_1} \leq x_k$, we get an inversion between x_{j_1} and x_k, because we assume that $y_i > y_j$ which yields $x_{i_1} > x_{j_1}$. Let $z_i, \ldots, z_{j-1}$ be all the elements from Z which appear between y_i and y_j in the original sequence. We know that there exists at least an inversion between $z_{\lfloor i+j \rfloor/2}$ and y_i or y_j. The inversion $(y_i, z_{\lfloor (i+j)/2 \rfloor})$ can be counted for two different pairs in Y, $(y_i, y_{i+2\lfloor (j-i)/2 \rfloor})$ and $(y_i, y_{i+1+2\lfloor (j-i)/2 \rfloor})$. Similarly, the inversion $(z_{\lfloor (i+j)/2 \rfloor, j})$ can be counted for two different pairs in Y. Taking into account that the inversions involving elements of Y and elements of

Z appear in X, but neither in Y nor Z, we have that $Inv(X) \geq Inv(Y)+Inv(Z)+Inv(Y)/2$. In a similar manner we obtain $Inv(X) \geq Inv(Y)+Inv(Z)+Inv(Z)/2$. Summing the two equations we obtain $Inv(X) \geq \frac{5}{4}(Inv(Y)+Inv(Z))$. □

Theorem 5. *GreedySort performs $O(N(1+\log(1+Inv(X)/N)))$ comparisons to sort a sequence X of size N, i.e. it is comparison optimal with respect to Inv.*

Proof. Similar to [17], we first prove the claimed bound for the upper levels of recursion where the total number of inversions is greater than $N/4$ and then prove that the total number of comparisons for the remaining levels is linear. Let $Inv_i(X)$ denote the total number of inversions in the subsequences at the i^{th} level of recursion. By Lemma 3, $Inv_i(X) \leq \left(\frac{4}{5}\right)^i Inv(X)$.

We want to find the first level ℓ of the recursion for which $\left(\frac{4}{5}\right)^\ell Inv(X) \leq \frac{N}{4}$, which yields $\ell = \left\lceil \frac{\log(4Inv(X)/N)}{\log(5/4)} \right\rceil$.

At each level of recursion GreedySort performs $O(N)$ comparisons. Therefore at the first ℓ levels of recursion the total number of comparisons performed is $O(\ell \cdot N) = O(N(1+\log(1+Inv(X)/N)))$. We now prove that the remaining levels perform a linear number of comparisons.

Let $|(X,i)|$ denote the total size of Ys and Zs at the ith level of recursion. As each element in Y and Z is obtained as a result of an inversion in the sequence X, we have $|(X,i)| \leq Inv_{i-1}(X)$. Using Lemma 3 we obtain: $|(X,\ell+i)| \leq Inv_{\ell+i-1}(X) \leq \left(\frac{4}{5}\right)^{i-1} \cdot \left(\frac{4}{5}\right)^\ell \cdot Inv(X) \leq \left(\frac{4}{5}\right)^{i-1} \frac{N}{4}$. Taking into account that the sum of the $|(X,\ell+i)|$s is $O(N)$ and that at each level $\ell+i$ we perform a linear number of comparisons with respect to $|(X,\ell+i)|$, it follows that the total number of comparisons performed at the lower levels of the recursion tree is $O(N)$. We conclude that GreedySort performs $O(N(1+\log(1+\frac{Inv}{N})))$ comparisons. □

We derive both cache-aware and cache-oblivious algorithms by using our greedy division protocol in both the cache-aware and the cache-oblivious GenericSort frameworks described in Sections 4 and 5. In both cases the division protocol considered does not identify the longest prefix of the input, but simply apply the greedy division protocol. We prove that these new algorithms, *cache-aware GreedySort* and *cache-oblivious GreedySort* achieve the I/O-optimality with respect to Inv under the tall cache assumption $M = \Omega(B^2)$.

Theorem 6. *Cache-aware GreedySort and cache-oblivious GreedySort are I/O-optimal with respect to Inv, provided that $M = \Omega(B^2)$.*

Proof. From Theorem 5 the average number of levels of recursion for an element is $O(1+\log(1+Inv/N))$. In Theorem 4 each element is charged $O(\frac{1}{B})$ I/Os for every $\Theta(\log \frac{M}{B})$ levels. This implies that cache-aware GreedySort performs $\Theta(\frac{N}{B}(1+\log_{\frac{M}{B}}(1+\frac{Inv}{N})))$ I/Os, which is optimal by Theorem 1. Similar observations apply to cache-oblivious GreedySort based on the proof of Lemma 2. □

References

1. A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.
2. L. Arge. External memory data structures. In J. Abello, P. M. Pardalos, and M. G. C. Resende, editors, *Handbook of Massive Data Sets*.
3. L. Arge, G. S. Brodal, and R. Fagerberg. Cache-oblivious data structures. In D. Mehta and S. Sahni, editors, *Handbook of Data Structures and Applications*, page 27. CRC Press, 2004.
4. L. Arge, M. Knudsen, and K. Larsen. A general lower bound on the I/O-complexity of comparison-based algorithms. In *Proc. of Workshop on Algorithms and Data Structures*, 1993.
5. M. Blum, R. W. Floyd, V. Pratt, R. L. Rivest, and R. E. Tarjan. Time bounds for selection. *J. Comput. Syst. Sci.*, 7:448–461, 1973.
6. G. S. Brodal. Cache-oblivious algorithms and data structures. In *Proc. 9th Scandinavian Workshop on Algorithm Theory*, Lecture Notes in Computer Science. Springer Verlag, Berlin, 2004.
7. G. S. Brodal and R. Fagerberg. Cache oblivious distribution sweeping. In *Proc. 29th International Colloquium on Automata, Languages, and Programming*, pages 426–438. Springer Verlag, Berlin, 2002.
8. G. S. Brodal and R. Fagerberg. On the limits of cache-obliviousness. In *Proc. 35th Annual ACM Symposium on Theory of Computing*, pages 307–315, 2003.
9. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, 2nd Edition*. MIT Press, 2001.
10. E. Demaine. Cache-oblivious algorithms and data structures. *Lecture Notes from the EEF Summer School on Massive Data Sets*, 2002.
11. V. Estivill-Castro and D. Wood. A new measure of presortedness. *Information and Computation*, 83(1):111–119, 1989.
12. V. Estivill-Castro and D. Wood. Practical adaptive sorting. In *Advances in Computing and Information - Proc. of the International Conference on Computing and Information*, pages 47–54. Springer-Verlag, 1991.
13. V. Estivill-Castro and D. Wood. A survey of adaptive sorting algorithms. *ACM Computing Surverys*, 24(4):441–475, 1992.
14. M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache oblivious algorithms. In *40th Ann. IEEE Symp. on Foundations of Computer Science*, pages 285–298, 1999.
15. L. J. Guibas, E. M. McCreight, M. F. Plass, and J. R. Roberts. A new representation of linear lists. In *Proc. 9th Ann. ACM Symp. on Theory of Computing*, pages 49–60, 1977.
16. D. E. Knuth. *The Art of Computer Programming. Vol 3, Sorting and searching*. Addison-Wesley, 1973.
17. C. Levcopoulos and O. Petersson. Splitsort – an adaptive sorting algorithm. *Information Processing Letters*, 39(1):205–211, 1991.
18. H. Manilla. Measures of presortedness and optimal sorting algorithms. *IEEE Trans. Comput.*, 34:318–325, 1985.
19. K. Mehlhorn. *Data structures and algorithms. Vol. 1, Sorting and searching*. Springer, 1984.
20. A. Pagh, R. Pagh, and M. Thorup. On adaptive integer sorting. In *Proc. 12th Annual European Symposium on Algorithms*, volume 3221, pages 556–567. 2004.
21. J. S. Vitter. External memory algorithms and data structures: Dealing with massive data. *ACM Computing Surveys*, 33(2):209–271, 2001.