External String Sorting: Faster and Cache-Oblivious

Rolf Fagerberg[⋆], Anna Pagh[⋆⋆], and Rasmus Pagh[⋆⋆]

Abstract. We give a randomized algorithm for sorting strings in external memory. For K binary strings comprising N words in total, our algorithm finds the sorted order and the longest common prefix sequence of the strings using $O(\frac{K}{B}\log_{M/B}(\frac{K}{M})\log(\frac{N}{K})+\frac{N}{B})$ I/Os. This bound is never worse than $O(\frac{K}{B}\log_{M/B}(\frac{K}{M})\log\log_{M/B}(\frac{K}{M})+\frac{N}{B})$ I/Os, and improves on the (deterministic) algorithm of Arge et al. *(On sorting strings in external memory, STOC '97).* The error probability of the algorithm can be chosen as $O(N^{-c})$ for any positive constant c. The algorithm even works in the cache-oblivious model under the tall cache assumption, i.e,, assuming $M > B^{1+\epsilon}$ for some $\epsilon > 0$. An implication of our result is improved construction algorithms for external memory string dictionaries.

1 Introduction

Data sets consisting partly or entirely of string data are common: Most database applications have strings as one of the data types used, and in some areas, such as bioinformatics, web retrieval, and word processing, string data is predominant. Additionally, strings form a general and fundamental data model of computer science, containing e.g. integers and multi-dimensional data as special cases.

In internal memory, sorting of strings is well understood: When the alphabet is comparison based, sorting K strings of total length N takes $\Theta(K\log K+N)$ time (see e.g. [8]). If the alphabet is the integers, then on a word-RAM the time is $\Theta(\mathrm{Sort}_{\mathrm{Int}}(K)+N)$, where $\mathrm{Sort}_{\mathrm{Int}}(K)$ is the time to sort K integers [3].

In external memory the situation is much less clear. Some upper bounds have been given [7], along with matching lower bounds in restricted models of computation. As noted in [7], the natural upper bound to hope for is $O(\frac{K}{B}\log_{M/B}(\frac{K}{M})+\frac{N}{B})$ I/Os, which is the sorting bound for K single characters plus the complexity of scanning the input. In this paper we show how to compute (using randomization) the sorted order in a number of I/Os that nearly matches this bound.

1.1 Models of Computation

Computers contain a hierarchy of memory levels, with large differences in access time. This makes the time for a memory access depend heavily on what is cur-

[⋆] University of Southern Denmark, Campusvej 55, 5230 Odense M, Denmark. Email: rolf@imada.sdu.dk

[⋆⋆] IT University of Copenhagen, Rued Langgaards Vej 7, 2300 København S, Denmark. Email: {annao,pagh}@itu.dk

rently the innermost level containing the data accessed. In algorithm analysis, the standard RAM (or von Neumann) model is unable to capture this, and external memory models were introduced to better model these effects. The model most commonly used for analyzing external memory algorithms is the two-level I/O-model [1], also called the External Memory model or the Disk Access model. The I/O-model approximates the memory hierarchy by modeling two levels, with the inner level having size M, the outer level having infinite size, and transfers between the levels taking place in blocks of B consecutive elements. The cost measure of an algorithm is the number of memory transfers, or I/Os, it makes.

The cache-oblivious model, introduced by Frigo et al. [17], elegantly generalizes the I/O-model to a multi-level memory model by a simple measure: the algorithm is not allowed to know the value of B and M. More precisely, a cache-oblivious algorithm is an algorithm formulated in the RAM model, but analyzed in the I/O-model, with an analysis valid for *any* value of B and M. Cache replacement is assumed to take place automatically by an optimal off-line cache replacement strategy. Since the analysis holds for any B and M, it holds for all levels simultaneously. See [17] for the full details of the cache-oblivious model.

Over the last two decades, a large body of results for the I/O-model has been produced, covering most areas of algorithmics. For the newer cache-oblivious model, introduced in 1999, already a sizable number of results exist. One of the fundamental facts in the I/O-model is that comparison-based sorting of N elements takes $\Theta(\mathrm{Sort}(N))$ I/Os [1], where $\mathrm{Sort}(N) = \frac{N}{B} \log_{M/B} \frac{N}{M}$. Also in the cache-oblivious model, sorting can be carried out in $\Theta(\mathrm{Sort}(N))$ I/Os, if one makes the so-called *tall cache* assumption $M \geq B^{1+\varepsilon}$ [10, 17]. This assumption has been shown to be necessary [11]. Another basic fact in the I/O-model is that permutation takes $\Theta(\min\{\mathrm{Sort}(N), N\})$, assuming that elements are indivisible [1].

The subject of this paper is sorting strings in external memory models. Below, we discuss existing results in this area. For a general overview of results for external memory, refer to the recent surveys [4, 21, 23, 24] for the I/O-model, and [6, 9, 13, 21] for the cache-oblivious model.

1.2 Previous Work

Arge et al. [7] were the first to study string sorting in external memory, introducing the size N of the input and the number K of strings as separate parameters. Note that the problem is at least as hard as sorting K words[1], and also requires at least N/B I/Os for reading the input. In internal memory, i.e., for $B = 1$, there are algorithms that meet this lower bound [3, 8]. However, it remains an open problem whether this is possible in external memory for general B.

Arge et al. give several algorithms obeying various indivisibility restrictions. The complexity of these algorithms depends on the number K_1 of strings of less than one block in length, and the number K_2 of strings of at least one block

[1] See Section 1.3 below for the model of computation.

in length. The total length of short and long strings is denoted N_1 and N_2, respectively. The fastest algorithm runs in

$$O(\min(K_1 \log_M K_1, \tfrac{N_1}{B} \log_{M/B}(\tfrac{N_1}{M})) + K_2 \log_M K_2 + \tfrac{N}{B}) \text{ I/Os.}$$

Under the tall cache assumption this simplifies to

$$O(\tfrac{N_1}{B} \log_M K_1 + K_2 \log_M K_2 + \tfrac{N_2}{B}) \text{ I/Os.}$$

The first term is the complexity of sorting the short strings using external merge sort. The second term states a logarithmic (base M) I/O cost per long string. The third term is the complexity of reading the long strings. Each of the three terms may be the dominant one. Assume for simplicity that all K strings have the same length. If they are short, the first term obviously dominates. If their length is between B and $B \log_M K$, the second term dominates. For longer strings, the third term dominates, i.e., sorting can be done in scanning complexity. Note that the upper bound is furthest from the lower bound for strings with a length around one block.

Arge et al. also consider "practical algorithms" whose complexity depends on the alphabet size. However, because of the implicit assumption that the alphabet has size $N^{\Theta(1)}$ (it is assumed that a character and a pointer uses the same space, within a constant factor), none of these algorithms are better than the above from a theoretical perspective.

Note that prior to our work there are no direct results on *cache-oblivious* string sorting, except the bound $O(\text{Sort}(N))$, which can be derived from suffix tree/array construction algorithms [14, 19].

1.3 Our results.

Before we state our results, we discuss the model and notation. First of all, we assume the input to be binary strings. This is no restriction in practice, since any finite alphabet can be encoded as binary strings such that replacing each character with its corresponding string preserves the ordering of strings. However, to facilitate a clear comparison with previous results, we will not count the length of strings in bits, but rather in *words* of $\Theta(\log N)$ bits. This is consistent with [7] which assumes that a character and a pointer uses the same space, within a constant factor. We will also assume that all strings have length at least one word, which again is consistent with [7]. We will adopt the notation from [7]:

$$\begin{aligned} K &= \# \text{ of strings to sort,} \\ N &= \text{total } \# \text{ of words in the K strings,} \\ M &= \# \text{ of words fitting in internal memory,} \\ B &= \# \text{ of words per disk block,} \end{aligned}$$

where $M < N$ and $1 < B \leq M/2$. We assume that the input sequence $x_1, \ldots, x_K$ is given in a form such that it can be read in $O(N/B)$ I/Os.

Secondly, we distinguish between standard *value-sorting*, which produces a sequence with the input items in sorted order, and the problem of finding the sorting permutation, i.e., producing a sequence of *references* to the items in sorted order (this is equivalent to what is sometimes referred to as *rank-sorting*, i.e., computing the rank of all input elements). For strings, the latter is often enough, as is e.g. the case for string dictionaries. The *sorting permutation* σ of the input sequence is the permutation such that $\sigma(i) = j$ if x_j has rank i in the sorted order. In this definition, the references to strings are their ranks in input order. If one instead as references wants pointers to the memory locations of the first characters of the strings, conversion between the two representations can be done in $O(\mathrm{Sort}(K) + N/B)$ I/Os by sorting and scanning. The latter representation is commonly called the suffix array in the case where the strings are the suffixes of some base string. Let $\mathrm{lcp}(x_i, x_j)$ be the longest common prefix of x_i and x_j. By the *lcp sequence* we denote the numbers $\mathrm{LCP}(i) = |\mathrm{lcp}(x_{\sigma(i)}, x_{\sigma(i+1)})|$, i.e., the lengths of the longest common prefixes between pairs of strings consecutive in sorted order. For the application of string dictionary construction, we will need this.

Our main result is a Monte Carlo type randomized, cache-oblivious algorithm which computes the sorting permutation and the lcp sequence. The output is the sequences $\sigma(1), \dots, \sigma(K)$ and $\mathrm{LCP}(1), \dots, \mathrm{LCP}(K-1)$.

Theorem 1. *Let c and ϵ be arbitrary positive constants, and assume that $M > B^{1+\epsilon}$. Then there is a randomized, cache-oblivious algorithm that, given K strings of N words in total, computes the sorting permutation and the lcp sequence in $O(\frac{K}{B}\log_{M/B}(\frac{K}{M})\log(\frac{N}{K}) + \frac{N}{B})$ I/Os, such that the result is correct with probability $1 - O(N^{-c})$.*

Note that the I/O bound above coincides with $\mathrm{Sort}(K)$ for strings of length $O(1)$ words, and is never worse than $O(\mathrm{Sort}(K)\log\log_{M/B}(\frac{K}{M}) + \frac{N}{B})$. (If $\frac{N}{K}$ exceeds $(\log_{M/B}(\frac{K}{M}))^{O(1)}$ then the N/B term will be dominant.) Thus, we have optimal dependence on N and are merely a doubly logarithmic factor away from $\mathrm{Sort}(K)$. We prove Theorem 1 in Section 2.

The String B-tree [15] is an external memory string dictionary, which allows (prefix) searches over a set of K strings in $O(\log_B K + P/B)$ I/Os, where P is the length of the search string. Constructing the String B-tree over a set of strings is as hard as finding the sorting permutation. Conversely, from the sorting permutation and the lcp sequence, the String B-tree over the strings can easily be built. Recently, a cache-oblivious string dictionary with the same searching complexity as String B-Trees has been given [12], and the same statement about construction applies to this structure. Hence, one important corollary of Theorem 1 is the following, shown in Section 3:

Corollary 1. *Let c and ϵ be arbitrary positive constants, and assume that $M > B^{1+\epsilon}$. Then there is a randomized, cache-oblivious algorithm that, given K strings of N words in total, constructs a String B-tree or a cache-oblivious string dictionary [12] over the strings in $O(\frac{K}{B}\log_{M/B}(\frac{K}{M})\log(\frac{N}{K}) + \frac{N}{B})$ I/Os, such that the result is correct with probability $1 - O(N^{-c})$.*

Again, this bound is never worse than $O(\mathrm{Sort}(K)\log\log_{M/B}(\frac{K}{M})+\frac{N}{B})$. If one wants not only the sorting sequence, but also the strings to appear in memory in sorted order, they must be permuted. In Section 4, we discuss methods for permuting strings based on knowledge of the sorting sequence. The methods are straightforward, but nevertheless show that also for the standard value-sorting problem (i.e. including permuting the strings), our main result leads to asymptotical improvements in complexity, albeit more modest than for finding the permutation sequence.

1.4 Other Related Work.

Sorting algorithms for the word RAM model have developed a lot in the last decade. The new RAM sorting algorithms take advantage of the bit representation of the strings or integers to be sorted, in order to beat the $\Omega(K\log K)$ lower bound for sorting K items using comparisons. The currently fastest sorting algorithm for K words runs in time $O(K\sqrt{\log\log K})$, expected [18]. This implies an external memory algorithm running in the same I/O bound, which is better than $O(\mathrm{Sort}(K))$ if K is sufficiently large ($K = M^{\omega(B)}$ is necessary).

Andersson and Nilsson [3] have shown how to reduce sorting of K strings of length N to sorting of $O(K)$ words, in $O(N)$ expected time. This means that the relation between string sorting and word sorting on a word RAM is very well understood. The currently fastest word sorting algorithm gives a bound of $O(K\sqrt{\log\log K}+N)$ expected time. Again, using this directly on external memory gives a bound better than other string sorting bounds (including those in the present paper) for certain extreme instances (very large sets of not too long strings).

If the length w of the machine words to be sorted is sufficiently large in terms of K, there exists an expected linear time word RAM sorting algorithm. Specifically, if $w > (\log K)^{2+\epsilon}$, for some constant $\epsilon > 0$, K words can be sorted in expected $O(K)$ time [2]. To understand the approach of the algorithm it is useful to think of the words as binary strings of length w. The key ingredient of the algorithm is a randomized signature technique that creates a set of "signature" strings having, with high probability, essentially the same trie structure (up to ordering of children of nodes) as the original set of strings. If the word length is large, the signatures can be made considerably shorter than the original strings, and after a constant number of recursive steps they can be sorted in $O(K)$ time. To sort the original strings, one essentially sorts the parts of the original strings that correspond to branching nodes in the trie of signatures.

Our algorithm is inspired by this, and uses the same basic approach as just described. However, the details of applying the technique to external memory strings are quite different. For example, it is easier for us to take advantage of the reduction in string size. This means that the best choice in our case is to use the signatures to decrease the string lengths by a constant factor at each recursive step, as opposed to the logarithmic factor used in [2]. Also, in the cache-oblivious model, it is not clear when to stop the recursion.

Using shorter strings to represent the essential information about longer ones was proposed already in [20]. Indeed, similar to the randomized signature scheme discussed above, the idea of Karp-Miller-Rosenberg labeling [20] is to divide the strings into substrings, and replace each substring with a shorter string. In particular, for each distinct substring one needs a unique shorter string to represent it. This can even be done such that the lexicographical order is preserved. The technique of [20] avoids randomization, but requires in each recursive step the sorting of the substrings that are to be replaced, which takes $O(\mathrm{Sort}(N))$ and hence will not improve on the known external memory upper bounds. The "practical algorithms" in [7] use this technique, but as stated above, these algorithms are asymptotically inferior to the best algorithm in [7].

2 Proof of main theorem

In this section we will prove Theorem 1. First we describe in Section 2.1 a recursive algorithm that finds the structure of the *unordered* trie of strings in the set S that is to be sorted. The algorithm is randomized and produces the correct trie with high probability. Each step of the recursion reduces the total length of the strings by a constant factor using a signature method. Section 2.2 then describes how to get from the unordered trie to the ordered trie, from which the sorting permutation and the lcp sequence can easily be found. The crux of the complexity bound is that at each recursive step, as well as at the final ordering step, only the (at most K) branching characters of the current trie are sorted, and the current set of strings is scanned. The use of randomization (hashing) allows the shorter strings of the next recursive step to be computed in scanning complexity (as opposed to the Karp-Miller-Rosenberg method), but also means that there is no relation between the ordering of the strings from different recursive levels. However, for unordered tries, equality of prefixes is all that matters.

Our algorithm considers the input as a sequence of strings $x_1, \ldots, x_K$ over an alphabet of size $\Theta(N^{2+c})$, by dividing the binary strings into chunks of $\lceil (2+c) \log N \rceil$ bits, where c is the positive constant of Theorem 1. Note that this means that the total length of all strings is $O(N)$ characters. Dividing into chunks may slightly increase the size of the strings, because the length is effectively rounded up to the next multiple of the chunk size. However, the increase is at most a constant factor. To simplify the description of our algorithm we make sure that S is *prefix free*, i.e., that no string in S is a prefix of another string in S. To ensure this we append to $x_1, \ldots, x_K$ special characters $\$_1, \ldots, \$_K$ that do not appear in the original strings. Extending the alphabet with K new characters may increase the representation size of each character by one bit, which is negligible.

2.1 Signature reduction.

We will now describe a recursive, cache-oblivious algorithm for finding the structure of the *blind trie* of the strings in S, i.e., the trie with all unary nodes removed

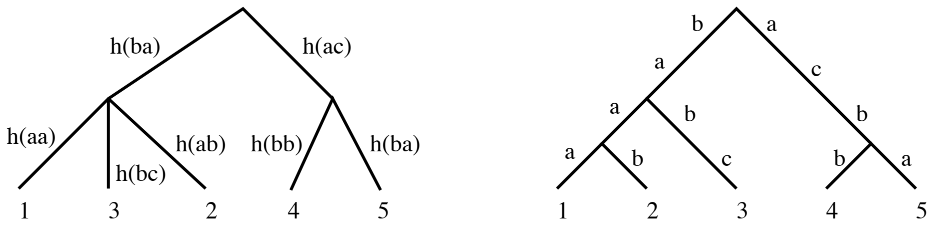

Fig. 1. We consider the set of strings $S = \{\texttt{baaa, baab, babc, acbb, acba}\}$. Left is $\mathcal{T}(S')$, where each character is a hash function value of two characters from S. Right is $\mathcal{T}(S)$, which can be computed from $\mathcal{T}(S')$ by considering each branching node in $\mathcal{T}(S')$ and its branching characters, each of which corresponds to a unique string of two characters from S.

(blind tries [15] are also known as compressed tries or Patricia tries [22]). Recall that for now, we are only concerned with computing the *unordered* blind trie $\mathcal{T}(S)$. We represent each node p as follows (where for brevity, a node is identified with the string represented by the path from the root to the node):

– A unique ID, which is a number in $\{1, \ldots, 2K\}$.
– The ID of its parent node q, which is the longest proper prefix of p in S, if any.
– The number i of a string $x_i \in S$ having p as a prefix (a *representative string*).
– Its *branching character*, i.e., the first character in p after q, and its position in p.

Our algorithm handles strings in S of length 1 separately, in order to be able to *not* recurse on these strings. Because no string is a prefix of another string, the strings of length 1 are leaf children of the root in $\mathcal{T}(S)$. Thus we may henceforth assume that S contains only the strings of length $\ell \geq 2$, and add the strings of length 1 to the trie at the end.

The sequence S' is derived from S by hashing pairs of consecutive characters to a single character, using a function chosen at random from a universal family of hash functions [16]. A string of ℓ characters in S will correspond to a string of $\lceil \ell/2 \rceil$ characters in S'. Since all strings have length at least 2, the total length of the strings in S' is at most $\frac{2}{3}N$, as desired. The set S' can be computed in a single scan of S, using $O(N/B)$ I/Os. We denote the strings of S' by $x'_1, \ldots, x'_K$ such that they correspond one by one to the strings $x_1, \ldots, x_K$ of S, in this order. The trie $\mathcal{T}(S')$ is computed recursively.

If there are no hash function collisions (i.e., no pair of distinct characters with the same hash function value), the longest common prefix of any two strings $x'_{i_1}, x'_{i_2} \in S'$ is of length $\lfloor |\mathrm{lcp}(x_{i_1}, x_{i_2})|/2 \rfloor$. Intuitively, this means that $\mathcal{T}(S')$ has the same structure as $\mathcal{T}(S)$, only "coarser". To get $\mathcal{T}(S)$ from $\mathcal{T}(S')$ we basically need to consider each node and its children, and introduce new branching nodes in this part of the trie by considering the (pairs of) characters of S corresponding to the branching characters. Figure 1 shows an example.

Assuming that the hash function has no collisions, $\mathcal{T}(S)$ can be computed from $\mathcal{T}(S')$ as follows:

1. Sort the nodes of $\mathcal{T}(S')$ according to the numbers of their representative strings in S', and secondly for each representative string according to the position of the branching character. Note that this can be done in a single sorting step (using a cache-oblivious sorting algorithm).
2. By scanning the strings of S in parallel with this sorted list, we can annotate each node p, having representative string x_i, with the two characters $c_{p,1}c_{p,2}$ from x_i that correspond to its branching character (their position can be computed from the position of the branching character in x'_i).
3. Sort the annotated nodes of $\mathcal{T}(S')$ according to the IDs of their parents, and for each parent ID according to $c_{p,1}$ (using a single sorting step).
4. We can now construct the nodes of $\mathcal{T}(S)$ by scanning this sorted list. Consider the children of a node p, occurring together in the list. There are two cases:
   (a) If all children have the same $c_{p,1}$ we can basically copy the structure of $\mathcal{T}(S')$. That is, we keep a node for each child p, with the same ID, parent ID, and representative string number as before, and with $c_{p,2}$ as branching character (its position can be computed as above).
   (b) If there are children having different $c_{p,1}$ we introduce a new node for each group of at least two children with the same $c_{p,1}$ (getting new IDs using a global internal memory counter). The branching character for such a node is $c_{p,1}$, the parent ID is that of p, and any representative string of a child can be used. Again, we keep a node for each child p with the same ID as in $\mathcal{T}(S')$. If no other child has the same $c_{p,1}$ the node keeps its parent ID, with branching character $c_{p,1}$. If two or more children have the same $c_{p,1}$, their parent is the new node with branching character $c_{p,1}$, and their branching characters are their $c_{p,2}$ characters.

Lemma 1. *The above algorithm uses $O(N/B)$ blocks of external space and $O(\frac{K}{B}\log_{M/B}(\frac{K}{M})\log(\frac{N}{K}) + N/B)$ I/Os. It computes $\mathcal{T}(S)$ correctly with probability $1 - O(N^{-c})$,*

Proof sketch. Since the length of the strings is geometrically decreasing during the recursion, the total length of all strings considered in the recursion is $O(N)$. This means that the space usage on external memory is $O(N/B)$ blocks, and that the number of pairs of characters hashed is $O(N)$. In particular, since the collision probability for any pair of inputs to the universal hash function is N^{-2-c}, we have that with probability $1-O(N^{-c})$ there are no two distinct pairs of characters that map to the same character (i.e., no hash function collisions). In this case, note that all sets of strings in the recursion are prefix free, as assumed by the algorithm. The argument that the trie is correct if there is no hash function collision is based on the fact that the longest common prefixes in S correspond to longest common prefixes of half the length (rounded down) in S'. Details will be given in the full version of this paper.

We now analyze the I/O complexity. At the ith level of the recursion the total length of the strings is bounded by $(\frac{2}{3})^i N$. In particular, for $i > \log_{3/2}(N/K)$ the maximum possible number of strings at each recursive level also starts to decrease geometrically (since the number of strings is bounded by the total length of the strings). Finally note that when the problem size reaches M, the rest of the recursion is completed in $O(M/B)$ I/Os.

Let $j = \lfloor \log_{3/2}(N/K) \rfloor$. We can bound the asymptotic number of I/Os used as follows:

$$\sum_{i=0}^{j} \left(\frac{K}{B} \log_{M/B}\left(\frac{K}{M}\right) + \frac{\left(\frac{2}{3}\right)^i N}{B} \right) + \sum_{i=j+1}^{\infty} \left(\frac{\left(\frac{2}{3}\right)^{i-j} K}{B} \log_{M/B}\left(\frac{\left(\frac{2}{3}\right)^{i-j} K}{M} \right) + \frac{\left(\frac{2}{3}\right)^i N}{B} \right)$$

$$= \left(\frac{K}{B} \log_{M/B}(\tfrac{K}{M}) \log(\tfrac{N}{K}) + \frac{N}{B} \right) + \left(\frac{K}{B} \log_{M/B}(\tfrac{K}{M}) + \frac{N}{B} \right) \quad .$$

The first term is the dominant one, and identical to the bound claimed. $\square$

2.2 Full algorithm.

We are now ready to describe our string sorting algorithm in its entirety. We start by finding $\mathcal{T}(S)$ using the algorithm of Section 2.1. What remains to find the sorting permutation is to order the children of each node according to their branching character and traverse the leaves from left to right. We do this by a reduction to *list ranking*, proceeding as follows:

1. Sort the nodes according to parent ID, and for each parent ID according to branching character, in a single sorting step.
2. We now construct a graph (which is a directed path) having two vertices, v^{in} and v^{out}, for each vertex v of $\mathcal{T}(S)$.
   (a) For a node with ID v having d (ordered) children with IDs $v_1, \ldots, v_d$ we construct the edges $(v^{\text{in}}, v_1^{\text{in}}), (v_1^{\text{out}}, v_2^{\text{in}}), \ldots, (v_{d-1}^{\text{out}}, v_d^{\text{in}}), (v_d^{\text{out}}, v^{\text{out}})$. We annotate each "horizontal" edge $(v_i^{\text{out}}, v_{i+1}^{\text{in}})$ by the length of the *lcp* of the representative strings of v_i and v_{i+1} (considered as bit strings). This number can be computed from the branching characters of v_i and v_{i+1} and their positions.
   (b) For a leaf node v we construct the edge $(v^{\text{in}}, v^{\text{out}})$, and annotate it with the number of its representative string.
3. Run the optimal cache-oblivious list ranking algorithm of [5] on the above graph to get the edges in the order they appear on the path.
4. Scan the list, and report the numbers annotated on the edges corresponding to the leaves in $\mathcal{T}(S)$ (giving the sorting permutation), and the numbers on the horizontal edges (giving the *lcp* sequence).

The work in this part of the algorithm is dominated by the sorting and list ranking steps, which run in $O(\frac{K}{B} \log_{M/B}(\frac{K}{M}) + N/B)$ I/Os. Again, we postpone the (straightforward) correctness argument to the full version of this paper. This concludes the proof sketch of Theorem 1.

3 Construction of External String Dictionaries

In this section, we prove Corollary 1. A String B-tree [15] is a form of B-tree over pointers to the strings. Each B-tree node contains pointers to $\Theta(B)$ strings, as well as a blind trie over these strings to guide the search through the node. The bottom level of the tree contains pointers (in sorted order) to all strings. These pointers are divided into groups of $\Theta(B)$ consecutive pointers, and a node is built on each group. For each node, the left-most and the right-most pointer are copied to the next level, and these constitute the set of pointers for this next level. Iterating this process defines all levels of the tree.

Building a blind trie on a set of strings is straightforward given the sequence of pointers to the strings in sorted order and the associated lcp sequence: insert the pointers as leaves of the trie in left-to-right order, while maintaining the right-most path of the trie in a stack. The insertion of a new leaf entails popping from the stack until the first node on the right-most path is met which has a string depth at most the length of the longest common prefix of the new leaf and its predecessor. The new leaf can now be inserted, possibly breaking an edge and creating a new internal node. The splitting characters of the two edges of the internal node can be read from the lcp sequence. The new internal node and the leaf is then pushed onto their stack.

For analysis, observe that once a node is popped from the stack, it leaves the right-most path. Hence, each node of the trie is pushed and popped once, for a total of $O(S)$ stack operations, where S is the number of strings represented by the blind trie. Since a stack implemented as an array is I/O-efficient by nature, the number of I/Os for building a blind trie is $O(S/B)$.

Hence, the number of I/Os for building the lowest level of the String B-tree is $O(K/B)$. Finding the pointers in sorted order and their corresponding lcp array for the next level is straightforward, using that the lcp value between any pairs of strings is the minimum of the lcp values for all intervening neighboring (in sorted order) pairs of strings. Hence, the lcp value of the left-most and right-most leaf in the trie in a String B-tree node can be found by scanning the relevant part of the lcp array. As the sizes of each level decreases by a factor $\Theta(B)$, building the entire String B-tree is dominated by the building of its lowest level.

For the cache-oblivious string dictionary [12], it is shown in [12] how to take a blind trie (given as an edge list) for a set of K strings, and build a cache-oblivious search structure for it, using $\mathrm{Sort}(K)$ I/Os. The construction algorithm works in the cache-oblivious model. Since an array-based stack is I/O efficient also in the cache-oblivious model, the blind trie can be built cache-obliviously by the method above. The ensuing search structure provides a cache-oblivious dictionary over the strings.

4 Bounds for Permuting Strings

In this section, we discuss methods for permuting the strings into sorted order in memory, based on knowledge of the sorting sequence. Our methods are straightforward, but nevertheless show that also for the standard sorting problem (i.e. including permuting the strings), our main result leads to asymptotical improvements (albeit modest) in complexity.

In the cache-aware case, knowledge of B allows us to follow Arge et al. [7], and divide the strings into short and long strings. We use their terminology K_1, K_2, N_1, N_2 described in Section 1.2. The long strings we permute by direct placement of the strings, based on preprocessing in $O(\mathrm{Sort}(K) + N/B)$ I/Os which calculates the position of each string. The short strings we permute by sorting, using the best algorithm from [7] for short strings. For simplicity of expression, we assume a tall cache. Then the complexity of our randomized sorting algorithm, followed by the permutation procedure above, is

$$O(\tfrac{K}{B} \log_M(K) \log(\tfrac{N}{K}) + \tfrac{N}{B} + \tfrac{N_1}{B} \log_M(K_1) + K_2) \,.$$

The bound holds for any choice of length threshold between short and long strings. For the threshold equal to B, it is easy to see that the bound improves on the bound of [7] for many parameter sets with long strings. For specific inputs, other thresholds may actually be better.

Turning to the cache-oblivious situation, we can also, on instances with long strings, improve the only existing bound of $O(\mathrm{Sort}(N))$. For simplicity assume a tall cache assumption of $M \geq B^2$. If $N \leq K^2$, we permute by sorting the words of the strings in $O(\mathrm{Sort}(N))$ as usual. Else, we place each string directly (using preprocessing as above). If $M > N$, everything is internal. Otherwise, $N/K \geq \sqrt{N} \geq \sqrt{M} \geq B$, so $N/B \geq K$, and we can afford one random I/O per string placed, leading to permutation in $O(\mathrm{Sort}(K) + N/B)$.

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. A. Andersson, T. Hagerup, S. Nilsson, and R. Raman. Sorting in linear time? *J. Comput. System Sci.*, 57(1):74–93, 1998.
3. A. Andersson and S. Nilsson. A new efficient radix sort. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science (FOCS '94)*, pages 714–721. IEEE Comput. Soc. Press, 1994.
4. L. Arge. External memory data structures. In J. Abello, P. M. Pardalos, and M. G. C. Resende, editors, *Handbook of Massive Data Sets*, pages 313–358. Kluwer Academic Publishers, 2002.
5. L. Arge, M. A. Bender, E. D. Demaine, B. Holland-Minkley, and J. I. Munro. Cache-oblivious priority queue and graph algorithm applications. In ACM, editor, *Proceedings of the 34th Annual ACM Symposium on Theory of Computing (STOC '02)*, pages 268–276. ACM Press, 2002.
6. L. Arge, G. S. Brodal, and R. Fagerberg. Cache-oblivious data structures. In D. Mehta and S. Sahni, editors, *Handbook on Data Structures and Applications*. CRC Press, 2005.
7. L. Arge, P. Ferragina, R. Grossi, and J. S. Vitter. On sorting strings in external memory (extended abstract). In ACM, editor, *Proceedings of the 29th Annual ACM*

Symposium on Theory of Computing (STOC '97), pages 540–548. ACM Press, 1997.

8. J. Bentley and R. Sedgewick. Fast algorithms for sorting and searching strings. In *Proc. 8th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 360–369, 1997.
9. G. S. Brodal. Cache-oblivious algorithms and data structures. In *Proc. 9th Scandinavian Workshop on Algorithm Theory*, volume 3111 of *Lecture Notes in Computer Science*, pages 3–13. Springer Verlag, Berlin, 2004.
10. G. S. Brodal and R. Fagerberg. Cache oblivious distribution sweeping. In *Proc. 29th International Colloquium on Automata, Languages, and Programming*, volume 2380 of *Lecture Notes in Computer Science*, pages 426–438. Springer Verlag, Berlin, 2002.
11. 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.
12. G. S. Brodal and R. Fagerberg. Cache-oblivious string dictionaries. In *Proceedings of the 17th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '06)*, 2006. To appear.
13. E. D. Demaine. Cache-oblivious data structures and algorithms. In *Proc. EFF summer school on massive data sets*, Lecture Notes in Computer Science. Springer, To appear.
14. M. Farach-Colton, P. Ferragina, and S. Muthukrishnan. On the sorting-complexity of suffix tree construction. *J. ACM*, 47(6):987–1011, 2000.
15. P. Ferragina and R. Grossi. The string B-tree: a new data structure for string search in external memory and its applications. *J. ACM*, 46(2):236–280, 1999.
16. M. L. Fredman and D. E. Willard. Trans-dichotomous algorithms for minimum spanning trees and shortest paths. *J. Comput. System Sci.*, 48(3):533–551, 1994.
17. M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache oblivious algorithms. In *40th Annual IEEE Symposium on Foundations of Computer Science*, pages 285–298. IEEE Computer Society Press, 1999.
18. Y. Han and M. Thorup. Integer sorting in $O(n\sqrt{\log\log n})$ expected time and linear space. In *Proceedings of the 43rd Annual Symposium on Foundations of Computer Science (FOCS '02)*, pages 135–144, 2002.
19. J. Kärkkäinen and P. Sanders. Simple linear work suffix array construction. In *Proc. 30th Int. Colloquium on Automata, Languages and Programming (ICALP)*, volume 2719 of *Lecture Notes in Computer Science*, pages 943–955. Springer Verlag, Berlin, 2003.
20. R. M. Karp, R. E. Miller, and A. L. Rosenberg. Rapid identification of repeated patterns in strings, trees and arrays. In *Proceedings of the 4th Annual ACM Symposium on Theory of Computing (STOC '72)*, pages 125–136, 1972.
21. U. Meyer, P. Sanders, and J. F. Sibeyn, editors. *Algorithms for Memory Hierarchies*, volume 2625 of *Lecture Notes in Computer Science*. Springer Verlag, Berlin, 2003.
22. D. R. Morrison. PATRICIA - practical algorithm to retrieve information coded in alphanumeric. *J. ACM*, 15(4):514–534, Oct. 1968.
23. J. S. Vitter. External memory algorithms and data structures: Dealing with MASSIVE data. *ACM Computing Surveys*, 33(2):209–271, 2001.
24. J. S. Vitter. Geometric and spatial data structures in external memory. In D. Mehta and S. Sahni, editors, *Handbook on Data Structures and Applications*. CRC Press, 2005.