Better External Memory Suffix Array Construction

Roman Dementiev, Juha Kärkkäinen, Jens Mehnert, Peter Sanders
MPI Informatik, Stuhlsatzenhausweg 85,
66123 Saarbrücken, Germany,
[dementiev,juha,jmehnert,sanders]@mpi-sb.mpg.de.

Abstract

Suffix arrays are a simple and powerful data structure for text processing that can be used for full text indexes, data compression, and many other applications in particular in bioinformatics. However, so far it looked prohibitive to build suffix arrays for huge inputs that do not fit into main memory. This paper presents design, analysis, implementation, and experimental evaluation of several new and improved algorithms for suffix array construction. The algorithms are asymptotically optimal in the worst case or on the average. Our implementation can construct suffix arrays for inputs of up to 4GByte in hours on a low cost machine where all previous implementations we are aware of would fail or take days.

We also present a simple and efficient external algorithm for checking whether an array of indexes is a suffix array.

As a tool of possible independent interest we present a systematic way to design, analyze, and implement *pipelined* algorithms.

1 Introduction

The suffix array [21, 12], a lexicographically sorted array of the suffixes of a string, has numerous applications, e.g., in string matching [21, 12], genome analysis [1] and text compression [6]. For example, one can use it as full text index: To find all occurrences of a pattern P in a text T do binary search in the suffix array of T, i.e., look for the interval of suffixes that have P as a prefix. A lot of effort has been devoted to efficient construction of suffix arrays, culminating recently in three direct linear time algorithms [16, 18, 19]. Considering all this, suffix arrays can therefore be viewed as a concept of at least equal importance to suffix trees. One of the linear time algorithms [16] is very simple and can also be adapted to obtain an optimal algorithm for external memory: Consider a machine with fast memory of size M and a secondary memory that can be accessed by I/Os to blocks of B consecutive words on each of D disks [25]. The DC3-algorithm [16] constructs a suffix array of a text T of length n using $\mathcal{O}(\mathrm{sort}(n))$ I/Os where $\mathrm{sort}(n) = \mathcal{O}\left(\frac{n}{DB}\log_{M/B}\frac{n}{M}\right)$ is the number of I/Os needed for sorting the characters of T which are integers in the range $1,\ldots, n$.

However, suffix arrays are still rarely used for processing huge inputs. Less powerful techniques like an index of all words appearing in a text are very simple, have favorable constant factors and can be implemented to work well with external memory for practical inputs. In contrast, the only previous external memory implementations of suffix array construction [7] are not only asymptotically suboptimal but also so slow that measurements could only be done for small inputs and artificially reduced internal memory size.

The main objective of the present paper is to narrow the gap between theory and practice by engineering algorithms for constructing suffix arrays that are at the same time asymptotically optimal and the best practical algorithms, and that can process really large inputs in realistic time. In the context of this paper, "engineering" includes algorithm design, theoretical analysis, careful implementation, and experiments with large, realistic inputs all working together to improve relevant constant factors, to understand realistic inputs, and to obtain fair comparisons between different algorithms.

Basic Concepts We use the shorthands $[i, j] = \{i, \ldots, j\}$ and $[i, j) = [i, j-1]$ for ranges of integers and extend to substrings as seen below. The input of our algorithms is an n character string $T = T[0]\cdots T[n-1] = T[0, n)$ of characters in the

alphabet $\Sigma = [1, n]$. The restriction to the alphabet $[1, n]$ is not a serious one. For a string T over any alphabet, we can first sort the characters of T, remove duplicates, assign a rank to each character, and construct a new string T' over the alphabet $[1, n]$ by renaming the characters of T with their ranks. Since the renaming is order preserving, the order of the suffixes does not change. A similar technique called *lexicographic naming* will play an important role in all of our algorithms where a string (e.g., a substring of T) is replaced by its rank in some set of strings.

There is a special character \$ which is smaller than any character in the alphabet. We use the convention that $T[i] = \$$ if $i \geq n$. $T_i = T[i, n)$ denotes the i-th suffix of T. The *suffix array* SA of T is a permutation of $[0, n)$ such that $T_{\mathrm{SA}[i]} < T_{\mathrm{SA}[j]}$ whenever $0 \leq i < j < n$. Let $\mathrm{lcp}(i, j)$ denote the longest common prefix length of $\mathrm{SA}[i]$ and $\mathrm{SA}[j]$ ($\mathrm{lcp}(i, j) = 0$ if $i < 0$ or $j \geq n$). Then we get the following derived quantities that can be used to characterize the "difficulty" of an input or that will turn out to play such a role in our analysis.

$$\mathrm{maxlcp} := \max_{0 \leq i < n} \mathrm{lcp}(i, i+1) \tag{1}$$

$$\overline{\mathrm{lcp}} := \frac{1}{n} \sum_{0 \leq i < n} \mathrm{lcp}(i, i+1) \tag{2}$$

$$\overleftrightarrow{\mathrm{lcp}}(i) := \max \left\{ \begin{matrix} \mathrm{lcp}(i-1, i), \\ \mathrm{lcp}(i, i+1) \end{matrix} \right\} \tag{3}$$

$$\log \overleftrightarrow{\mathrm{lcp}} := \frac{1}{n} \sum_{0 \leq i < n} \log(1 + \overleftrightarrow{\mathrm{lcp}}(i)) \tag{4}$$

The I/O model [25] assumes a machine with fast memory of size M words and a secondary memory that can be accessed by I/Os to blocks of B consecutive words on each of D disks [25]. Our algorithms use words of size $\lceil \log n \rceil$ bits for inputs of size n. Sometimes it is assumed that an additional bit can be squeezed in somewhere. We express all our I/O complexities in terms of the shorthands $\mathrm{scan}(x) = \lceil x/DB \rceil$ for sequentially reading or writing x words and $\mathrm{sort}(x) \approx \frac{2n}{DB} \left\lceil \log_{M/B} \frac{n}{M} \right\rceil$ for sorting x words of data (not counting the $2\mathrm{scan}(x)$ I/Os for reading the input and writing the output).

Our algorithms are described using high level Pascal like pseudocode mixed with mathematical notation. The scope of control structures is determined by indentation. We extend set notation to sequences in the obvious way. For example $[i : i \text{ is prime}] = \langle 2, 3, 5, 7, 11, 13, \ldots \rangle$ in that order.

Overview: In Section 2 we present the *doubling algorithm* [3, 7] for suffix array construction that has I/O complexity $\mathcal{O}(\mathrm{sort}(n \log \mathrm{maxlcp}))$. This algorithm sorts strings of size 2^k in the k-th iteration. Our variant already yields some small optimization opportunities.

Using this simple algorithm as an introductory example, Section 3 then introduces the technique of *pipelined* processing of sequences in a systematic way which saves a factor at least two in I/Os for many external algorithms and is supported by our external memory library STXXL . The main technical result of this section is a theorem that allows easy analysis of the I/O complexity of pipelined algorithms. This theorem is also applied to more sophisticated construction algorithms presented in the subsequent sections.

Section 4 gives a simple and efficient way to *discard* suffixes from further iterations of the doubling algorithm whose position in the suffix array is already known. This leads to an algorithm with I/O complexity $\mathcal{O}(\mathrm{sort}(n \log \overleftrightarrow{\mathrm{lcp}}))$ improving on a previous discarding algorithm with I/O complexity $\mathcal{O}(\mathrm{sort}(n \log \overleftrightarrow{\mathrm{lcp}}) + \mathrm{scan}(n \log \mathrm{maxlcp}))$ [7]. A further constant factor is gained in Section 5 by considering a generalization of the doubling technique that sorts strings of size a^k in iteration k. The best multiplication factor is four (*quadrupling*) or five. A pipelined optimal algorithm with I/O complexity $\mathcal{O}(\mathrm{sort}(n))$ in Section 6 concludes our sequence of suffix array construction algorithms.

A useful tool for testing our implementations was a fast and simple external memory checker for suffix arrays described in Section 7.

In Section 8 we report on extensive experiments using synthetic difficult inputs, the human genome, English books, web-pages, and program source code using inputs of up to 4 GByte on a low cost machine. The theoretically optimal algorithm turns out to be the winner closely followed by quadrupling with discarding.

Section 9 summarizes the overall results and discusses how even larger suffix arrays could be build. The appendix contains further details that will be part of the full paper.

More Related Work The first I/O optimal algorithm for suffix array construction [11] is based on suffix tree construction and introduced the basic divide-and-conquer approach that is also used by

DC3. However, the algorithm from [11] is so complicated that an implementation looks not promising.

There is an extensive implementation study for external suffix array construction by Crauser and Ferragina [7]. They implement several nonpipelined variants of the doubling algorithm [3] including one that discards unique suffixes. However, this variant of discarding still needs to scan all unique tuples in each iteration. With our analysis, one would get $\mathcal{O}(\text{sort}(n) \log \overrightarrow{\text{lcp}} + \text{scan}(n) \log \text{maxlcp})$ I/Os. Our discarding algorithm eliminates the second term that dominates the I/O volume for many inputs. Interestingly, an algorithm that fares very well in the study of [7] is the GBS-algorithm [12] that takes $\mathcal{O}\left(\frac{N}{M}\text{scan}(n)\right)$ I/Os in the *best case* and has dismal worst case performance[1]. In iteration i, the GBS-algorithm sorts the suffixes T_i for $i \in [in, (i+1)n)$ and then merges them with the previously sorted suffixes. The GBS-algorithm can have favourable I/O volume if N/M is a small constant. We have not implemented this algorithm not only because more scalable algorithms are more interesting but also because all our algorithmic improvements (pipelining, discarding, quadrupling, the DC3-algorithm) add to a dramatic reduction in I/O volume and are not applicable to the GBS-algorithm. Hence it is predictable that the range where the GBS-algorithm is interesting would get much smaller. Moreover, the GBS-algorithm needs a local suffix array search for each suffix scanned so that it is quite expensive with respect to internal work. Our system (multiple modern disks controlled by a performance oriented library [9]) supports disk I/O at a speed up to one third of its memory bandwidth [10] so that the high internal cost makes the GBS-algorithm even more questionable for the present study. Nevertheless it should be kept in mind that the GBS-algorithm might be interesting for small inputs and fast machines with slow I/O.

There has been considerable interest in space efficient internal memory algorithms for constructing suffix arrays [22, 5] and even more compact full-text indexes [20, 13, 14]. We view this as an indication that internal memory is too expensive for the big suffix arrays one would like to build. Going to external memory can be viewed as an alternative and more scalable solution to this problem. Once this step is made, space consumption is less of an issue because disk space is two orders of magnitude cheaper than RAM.

The biggest suffix array computations we are aware of are for the human genome [23, 20]. One [20] computes the compressed suffix array on PC with 3GByte of memory in 21 h. Compressed suffix arrays work well in this case (they need only 2 GByte of space) because the small alphabet size present in genomic information enables efficient compression. The other implementation [23] uses a supercomputer with 64 GByte of memory and needs 7 hours. Our algorithms have comparable speed using external memory.

Pipelining to reduce I/Os is well known technique in executing database queries [24]. However, previous algorithm libraries for external memory [4, 8] do not support it. We decided quite early in the design of our library STXXL [9] that we wanted to remove this deficit. Since suffix array construction can profit immensely from pipelining and since the different algorithms give a rich set of examples, we decided to use this application as a test bed for a prototype implementation of pipelining.

2 A Doubling Algorithm

Figure 1 gives pseudocode for the doubling algorithm. The basic idea is to replace characters $T[i]$ of the input by *lexicographic names* that respect the lexicographic order of the length 2^k substring $T[i, i+2^k)$ in the k-th iteration. In contrast to previous external implementation of this algoirthm, our formulation never actually builds the resulting string of names. Rather, it manipulates a sequence P of pairs where each character c is tagged with its position i in the input. To obtain names for the next iteration $k+1$, the names for $T[i, i+2^k)$ and $T[i+2^k, i+2^{k+1})$ together with the position i are stored in a sequence S and sorted. The new names can now be obtained by scanning this sequence and comparing adjacent tuples. Sequence S can be build using consecutive elements of P if we sort P using they pair $(i \bmod 2^k, i \operatorname{div} 2^k)$.[2] Previous formulations of the algorithm use i as a sorting criterion and therefore have to access elements that are 2^k characters apart. Our approach saves I/Os and simplifies the pipelining

[1] There is also an variant of the GBS-algorithm that gives the best case bound in the worst case [7]. But this algorithm needs a constant factor more passes over the input and hence might be slower in practice.

[2] $(i \bmod 2^k, i \operatorname{div} 2^k)$ can also be computed using a single left rotation by k-bits of the binary representation of i.

```
Function doubling(T)
    S:= [((T[i], T[i + 1]), i) : i ∈ [0, n)]                    (0)
    for k := 1 to ⌈log n⌉ do
        sort S                                                   (1)
        P:= name(S)                                              (2)
        invariant ∀(c, i) ∈ P :
            c is a lexicographic name for T[i, i + 2^k)
        if the names in P are unique then
            return [i : (c, i) ∈ P]                              (3)
        sort P by (i mod 2^k, i div 2^k))                        (4)
        S:= ⟨((c, c'), i) : j ∈ [0, n),                          (5)
                (c, i) = P[j], (c', i + 2^k) = P[j + 1]⟩

Function name(S : Sequence of Pair)
    q:= r:= 0;   (ℓ, ℓ'):= ($, $)
    result:= ⟨⟩
    foreach ((c, c'), i) ∈ S do
        q++
        if (c, c') ≠ (ℓ, ℓ') then r:= q;   (ℓ, ℓ'):= (c, c')
        append (r, i) to result
    return result
```

Figure 1: The doubling algorithm.

optimization described in Section 3.

The algorithm performs a constant number of sorting and scanning operations for sequences of size n in each iteration. The number of iterations is determined by the logarithm of the longest common prefix.

Theorem 1. *The doubling algorithm computes a suffix array using $\mathcal{O}(\text{sort}(n)\lceil\log \text{maxlcp}\rceil)$ I/Os.*

3 Pipelining

The I/O volume of the doubling algorithm from Figure 1 can be reduced significantly by observing that rather than writing the sequence S to external memory, we can directly feed it to the sorter in Line (1). Similarly, the sorted tuples need not be written but can be directly fed into the naming procedure in Line (2) which can in turn forward it to the sorter in Line (4). The result of this sorting operation need not be written but can directly yield tuples of S that can be fed into the next iteration of the doubling algorithm. Appendix A gives a simplified analysis of this example for *pipelining*.

Let us discuss a more systematic model: The computations in many external memory algorithms can be viewed as a data flow through a directed acyclic graph $G = (V = F \cup S \cup R, E)$. The *file nodes* F represent data that has to be stored physically on disk. When a file node $f \in F$ is accessed we need a buffer of size $b(f) = \Omega(BD)$. The *streaming nodes* $s \in S$ read zero, one or several sequences and output zero, one or several new sequences using internal buffers of size $b(s)$.[3] The *Sorting nodes* $r \in R$ read a sequence and output it in sorted order. Sorting nodes have a buffer requirement of $b(r) = \Theta(M)$ and outdegree one[4]. Edges are labeled with the number of machine words $w(e)$ flowing between two nodes. In the proof of Theorem 3 you find the flow graph for the pipelined doubling algorithm. We will see somewhat more complicated graph in Sections 6 and 4. The following theorem (proven in Appendix B) gives necessary and sufficient conditions for an I/O efficient execution of such a data flow graph. Moreover, it shows that streaming computations can be scheduled completely systematically in an I/O efficient way.

Theorem 2. *The computations of a data flow graph $G = (V = F \cup S \cup R, E)$ with edge flows $w : E \to \mathbb{R}_+$ and buffer requirements $b : V \to \mathbb{R}_+$ can be executed using*

$$\sum_{e \in E \cap (F \times V \cup V \times F)} \text{scan}(w(e)) + \sum_{e \in E \cap (V \times R)} \text{sort}(w(e)) \qquad (5)$$

I/Os iff the following conditions are fulfilled. Consider the graph G' which is a copy of G except that edges between streaming nodes are replaced by bidirected edges. The strongly connected components (SCCs) of G' are required to be either single file nodes, single sorting nodes, or sets of streaming nodes. The total buffer requirement of each SCC C of streaming nodes plus the buffer requirements of the nodes directly connected to C has to be bounded by the internal memory size M.

Theorem 2 can be used to design and analyze pipelined external memory algorithms in a systematic way. All we have to do is to give a data flow graph that fulfills the requirements and we can then read off the I/O complexity. Using the relations $a \cdot \text{scan}(x) = \text{scan}(a \cdot x) + \mathcal{O}(1)$ and $a \cdot \text{sort}(x) \leq$

[3] Streaming nodes may cause additional I/Os for internal processing, e.g., for large FIFO queues or priority queues. These I/Os are not counted in our analysis.

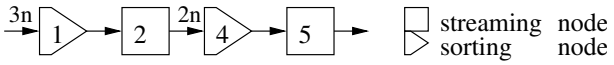
[4] We could allow additional outgoing edges at an I/O cost n/DB. However, this would mean to perform the last phase of the sorting algorithm several times.

$\text{sort}(a \cdot x) + \mathcal{O}(1)$, we can represent the result in the form $\text{scan}(x) + \text{sort}(y) + \mathcal{O}(1)$, i.e., we can characterize the complexity in terms of the *sorting volume* x and the scanning volume y. One could further evaluate this function by plugging in the I/O complexity of a particular sorting algoithm (e.g., $\approx 2x/DB$ for $x \ll M^2/DB$ and $M \gg DB$) but this may not be desirable because we loose information. In particular, scanning implies less internal work and can usually be implemented using *bulk I/Os* in the sense of [7] (we then need larger buffers $b(v)$ for file nodes) whereas sorting requires many random accesses for information theoretic reasons [2].

Now we apply Theorem 2 to the doubling algorithm:

Theorem 3. *The doubling algorithm from Figure 1 can be implemented to run using* $\text{sort}(5n)\,\lceil \log(1 + \text{maxlcp}) \rceil + \mathcal{O}(\text{scan}(n))$ *I/Os.*

Proof. The following flow graph shows that each iteration can be implemented using $\text{sort}(2n) + \text{sort}(3n) \leq \text{sort}(5n)$ I/Os. The numbers refer to the line numbers in Figure 1



After $\lceil \log(1 + \text{maxlcp}) \rceil$ iterations, the algorithm finishes. The $\mathcal{O}(\text{sort}(n))$ term accounts for the I/Os needed in Line 0 and for computing the final result. Note that there is a small technicality here: Although naming can find out "for free" whether all names are unique, the result is known only when naming finishes. However, at this time, the first phase of the sorting step in Line 4 has also finished and has already incurred some I/Os. Moreover, the convenient arrangement of the pairs in P is destroyed now. However we can then abort the sorting process, undo the wrong sorting, and compute the correct output. □

In STXXL the data flow nodes are implemented as objects with an interface similar to the STL *input* iterators [9]. A node reads data from input nodes using their `*` operators. With help of their preincrement operators a node proceeds to the next elements of the input sequences. The interface also defines an `empty()` function which signals the end of the sequence. After creating all node objects, the computation starts in a "lazy" fashion, first trying to evaluate the result of the topologically latest node. The node reads its input nodes element by element. Those nodes continue in the same mode, pulling the inputs needed to produce an output element. The process terminates when the result of the topologically latest node is computed. To support nodes with more than one output, STXXL exposes an interface where a node generates output accessible not only via the `*` operator but a node can also *push* an output element to output nodes.

The library already offers basic generic classes which implement the functionality of sorting, file, and streaming nodes.

4 Discarding Unique Suffixes

The procedure *name* in Figure 1 assigns a rank to each suffix as one plus the number of suffixes with strickly smaller prefix of length 2^k. If a suffix has a unique prefix of length 2^k, the rank assigned to it will not change in later iterations. The idea of discarding is to remove the pairs representing such finished suffixes from P and S, thus reducing the work and I/O in later iterations. This approach is particularly effective when $\log \overleftrightarrow{\text{lcp}} \ll \log \text{maxlcp}$, since while the algorithm still performs $\lceil \log \text{maxlcp} \rceil$ iterations, an average suffix is involved in only $\log \overleftrightarrow{\text{lcp}}$ of them.

There are two places in the algorithm in Figure 1 that are disturbed by discarding. The first one is in function *name*, where the rank can no more be computed by simple counting. The solution is to take the rank from the previous iteration (the number suffixes with strickly smaller prefix of length 2^{k-1}) and add to it the number of suffixes with the same prefix of length 2^{k-1} but smaller prefix of length 2^k [7]. The modified function *name2* is given in Figure 3. Note that *name2* cannot be used in the first iteration when no ranks have been computed yet.

The second problem with discarding is on Line (5) in Figure 1, where ranks of discarded suffixes may be needed as the component c' in S. As a correction, the discarded suffixes are included when computing S (a scan), but excluded during the rest of the algorithm (including all the sorting steps). Up to this point, the algorithm corresponds to the one in [7]. As an additional optimization, we will identify suffixes that are not needed even in the computation of S and store them separately to wait until the end of the algorithm. The rule to identify these fully discarded suffixes is simple: if a rank was not used in iteration k as a component of S, it will not be used in later iterations either. Figure 3 gives the final algo-

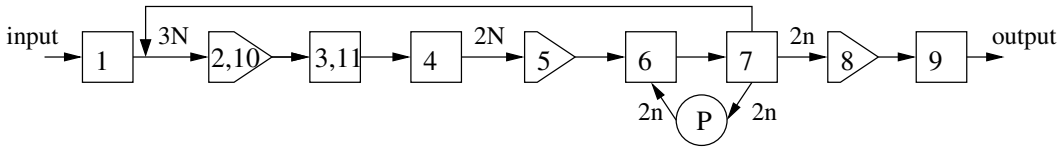


Figure 2: Data flow graph for the *doubling + discarding*. The numbers refer to line numbers in Figure 3. The edge weights are sums over the whole execution with $N = n \log \overleftrightarrow{\text{lcp}}$

rithm. A slightly different algorithm with the same asymptotic complexity is described in [15].

Function *doubling + discarding*(T)
- S:= $[((T[i], T[i+1]), i) : i \in [0, n)]$ (1)
- sort S (2)
- U:= *name*(S) // undiscarded (3)
- P:= $\langle\rangle$ // partially discarded
- F:= $\langle\rangle$ // fully discarded
- **for** $k := 1$ **to** $\lceil \log n \rceil$ **do**
  - mark unique names in U (4)
  - sort U by (i **mod** $2^k, i$ **div** 2^k) (5)
  - merge P into U; P:= $\langle\rangle$ (6)
  - S:= $\langle\rangle$; *discard*:= `1`
  - **foreach** $(c, i) \in U$ **do** (7)
    - **if** c is unique **then**
      - **if** *discard* = `1` **then**
        - append (c, i) to F
      - **else** append (c, i) to P
      - *discard*:= `1`
    - **else**
      - let (c', i') be the next pair in U
      - append $((c, c'), i)$ to S
      - *discard*:= `0`
  - **if** $S = \emptyset$ **then**
    - sort F by first component (8)
    - **return** $[i : (c, i) \in F]$ (9)
  - sort S (10)
  - U:= *name2*(S) (11)

Function *name2*(S : *Sequence* **of** *Pair*)
- q:= r:= 0; (ℓ, ℓ'):= (\$, \$)
- *result*:= $\langle\rangle$
- **foreach** $((c, c'), i) \in S$ **do**
  - **if** $c \neq \ell$ **then** q:= r:= 0; (ℓ, ℓ'):= (c, c')
  - **else if** $c' \neq \ell'$ **then** r:= q; ℓ':= c'
  - append $(c + r, i)$ to *result*
  - q++
- **return** *result*

Figure 3: The doubling with discarding algorithm.

Theorem 4. *Doubling with discarding can be implemented to run using* $\text{sort}(5n \log \overleftrightarrow{\text{lcp}}) + \mathcal{O}(\text{sort}(n))$ *I/Os.*

Proof. We prove the theorem by showing that the total amount of data in the different steps of the algorithm over the whole execution is as in the data flow graph in Figure 2. The nontrivial points are that $N = n \log \overleftrightarrow{\text{lcp}}$ tuples are processed in all sorting steps together and that at most n tuples are written to P. The former follows from the fact that a suffix i is involved in the sorting steps as long as it has a non-unique rank, which happens in exactly $\lceil \log(1 + \overleftrightarrow{\text{lcp}}(i)) \rceil$ iterations. To show the latter, we note that a tuple (c, i) is written to P in iteration k only if the previous tuple $(c', i - 2^k)$ was not unique. That previous tuple will become unique in the next iteration, because it is represented by $((c', c), i + 2^k)$ in S. Since each tuple turns unique only once, the total number of tuples written to P is at most n. □

5 From Doubling to a-Tupling

It is straightforward to generalize the doubling algorithms from Figures 1 and 3 so that it maintains the invariant that in iteration k, lexicographic names represent strings of length a^k: just gather a names from the last iteration that are a^{k-1} characters apart. Sort and name as before.

Theorem 5. *The a-tupling algorithm can be implemented to run using*

$$\text{sort}(\frac{a+3}{\log a} n) \log \text{maxlcp} + \mathcal{O}(\text{sort}(n)) \quad \textit{or}$$

$$\text{sort}(\frac{a+3}{\log a} n) \log \overleftrightarrow{\text{lcp}} + \mathcal{O}(\text{sort}(n))$$

I/Os without or with discarding respectively.

We get a tradeoff between higher cost for each iteration and a smaller number of iterations that is determined by the ratio $\frac{a+3}{\log a}$. Evaluating this expression we get the optimum for $a = 5$. But the value for

$a = 4$ is only 1.5 % worse, needs less memory, and calculations are much easier because four is a power two. Hence, we choose $a = 4$ for our implementation of the a-tupling algorithm. This *quadrupling* algorithm needs 30 % less I/Os than doubling.

6 A Pipelined I/O-Optimal Algorithm

Function *DC3*(T)
S:= $[((T[i, i+2]), i) : i \in [0, n), i \bmod 3 \neq 0]$ (1)
sort S by the first component (2)
P:= $name(S)$ (3)
if the names in P are not unique **then**
sort the $(i, r) \in P$ by $(i \bmod 3, i \operatorname{div} 3)$ (4)
SA^{12}:= $DC3([c : (c, i) \in P])$ (5)
P:= $\left[(j + 1, \mathrm{SA}^{12}[j]) : j \in [0, 2n/3)\right]$ (6)
sort P by the second component (7)
S_0:= $\langle (T[i], T[i+1], c', c'', i) :$ (8)
$i \bmod 3 = 0, (c', i+1), (c'', i+2) \in P\rangle$
S_1:= $\langle (c, T[i], c', i) :$ (9)
$i \bmod 3 = 1, (c, i), (c', i+1) \in P\rangle$
S_2:= $\langle (c, T[i], T[i+1], c'', i) :$ (10)
$i \bmod 3 = 2, (c, i), (c'', i+2) \in P\rangle$
sort S_0 by components 1,3 (11)
sort S_1 and S_2 by component 1 (12)
S:= $merge(S_0, S_1, S_2)$ comparison function: (13)
$(t, t', c', c'', i) \in S_0 \leq (d, u, d', j) \in S_1$
$\Leftrightarrow (t, c') \leq (u, d')$
$(t, t', c', c'', i) \in S_0 \leq (d, u, u', d'', j) \in S_2$
$\Leftrightarrow (t, t', c'') \leq (u, u', d'')$
$(c, t, c', i) \in S_1 \leq (d, u, u', d'', j) \in S_2$
$\Leftrightarrow c \leq d$
return [last component of $s : s \in S$] (14)

Figure 4: The DC3-algorithm.

The following three step algorithm outlines a linear time algorithm for suffix array construction [16]:

1. Construct the suffix array of the suffixes starting at positions $i \bmod 3 \neq 0$. This is done by reduction to the suffix array construction of a string of two thirds the length, which is solved recursively.
2. Construct the suffix array of the remaining suffixes using the result of the first step.
3. Merge the two suffix arrays into one.

Figure 4 gives pseudocode for an external implementation of this algorithm and Figure 5 gives a data flow graph that allows pipelined execution. Step 1 is implemented by Lines (1)–(6) and starts out quite similar to the tripling (3-tupling) algorithm described in Section 5. The main difference is that triples are only obtained for two thirds of the suffixes and that we use recursion to find lexicographic names that *exactly* characterize the relative order of these *sample suffixes*. As a preparation for the Steps 2 and 3, in lines (7)–(10) these sample names are used to annotate each suffix position i with enough information to determine its global rank. More precisely, at most two sample names and the first one or two characters suffice to completely determine the rank of a suffix. This information can be obtained I/O efficiently by simultaneously scanning the input and the names of the sample suffixes sorted by their position in the input. With this information, Step 2 reduces to sorting suffixes T_i with $i \bmod 3 = 0$ by their first character and the name for T_{i+1} in the sample (Line 11). Line (12) reconstructs the order of the mod-2 suffixes and mod-3 suffixes. Line (13) implements Step 3 by ordinary comparison based merging. The slight complication is the comparison function. There are three cases:

- A mod-0 suffix T_i can be compared with a mod-1 suffix T_j by looking at at the first characters and the names for T_{i+1} and T_{j+1} in the sample respectively.
- For a comparison between a mod-0 suffix T_i and a mod-2 suffix T_j the above technique does not work since T_{j+1} is not in the sample. However, both T_{i+2} and T_{j+2} are in the sample so that it suffices to look at the first two characters and the names of T_{i+2} and T_{j+2} respectively.
- Mod-1 suffixes and Mod-2 suffixes can be compared by looking at their names in the sample.

The resulting data flow graph is large but fairly straightforward except for the file node which stores a copy of input stream T. The problem is that the input is needed twice. First, Line 2 uses it for generating the sample and later, the node implementing Lines (8)–(10) scans it simultaneously with the names of the sample suffixes. It is not possible to pipeline both scans because we would violate the requirement of Theorem 2 that edges between streaming nodes must not cross sorting nodes. This problem

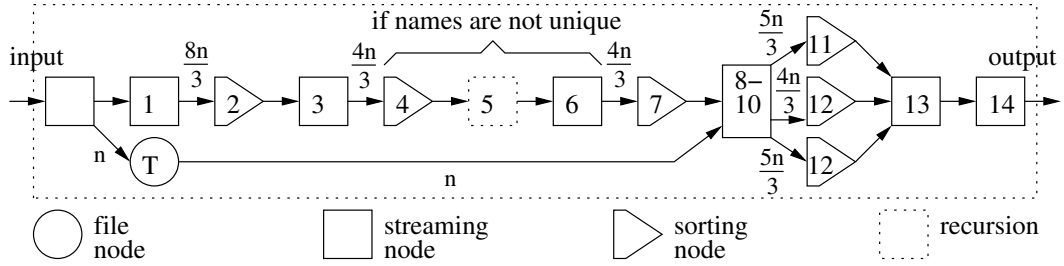


Figure 5: Data flow graphs for the DC3-algorithm. The numbers refer to line numbers in Figure 4

can be solved by writing a temporary copy of the input stream. Note that this is still cheaper than using a file representation for the input since this would mean that this file is read twice. We are now ready to analyze the I/O complexity of the algorithm.

Theorem 6. *The doubling algorithm from Figure 4 can be implemented to run using* $\text{sort}(30n)+\text{scan}(6n)$ *I/Os.*

Proof. Let $V(n)$ denote the number of I/Os for the external DC3 algorithm. Using Theorem 2 and the data flow diagram from Figure 5 we can conclude that

$$\begin{aligned} V(n) &\leq \text{sort}((\tfrac{8}{3}+\tfrac{4}{3}+\tfrac{4}{3}+\tfrac{5}{3}+\tfrac{3}{3}+\tfrac{5}{3})n) \\ &\quad + \text{scan}(2n) + V(\frac{2}{3}n) \\ &= \text{sort}(10n) + \text{scan}(2n) + V(\tfrac{2}{3}n) \end{aligned}$$

This recurrence has the solution $V(n) \leq 3(\text{sort}(10n)+\text{scan}(2n)) \leq \text{sort}(30n)+\text{scan}(6n)$. Note that the data flow diagram assumes that the input is a data stream into the procedure call. However, we get the same complexity if the original input is a file. In that case, we have to read the input once but we save writing it to the local file node T. □

7 A Checker

To ensure the correctness of our algorithms we have designed and implemented a simple and fast suffix array checker. It is given in Figure 6 and is based on the following result.

Lemma 1 ([5]). *An array $SA[0,n)$ is the suffix array of a text T iff the following conditions are satisfied:*

1. *SA contains a permutation of $[0,n)$.*
2. *Let r_i be the rank of the suffix S_i according to the suffix array. For all i,j, $r_i \leq r_j \Leftrightarrow (T[i], r_{i+1}) \leq (T[j], r_{j+1})$.*

Function *Checker*(SA, T)
P:= $[(SA[i], i+1) : i \in [0,n)]$ (1)
sort P by the first component (2)
if $[i : (i,r) \in S] \neq [0,n)$ **then return** false
S:= $[(r, (T[i], r')) : i \in [0,n),$ (3)
$(i,r) = P[i], (i+1, r') = P[i+1]]$
sort S by first component (4)
if $[(c,r') : (r,(c,r')) \in S]$ is sorted (5)
then return true **else return** false

Figure 6: The suffix array checker

Proof. The conditions are clearly necessary. To show sufficiency, assume that the suffix array contains exactly permutation of $[0,n)$ but in wrong order. Let S_i and S_j be a pair of wrongly ordered suffixes, say $S_i > S_j$ but $r_i < r_j$, that maximizes $i+j$. The second conditions is violated if $T[i] > T[j]$. Otherwise, we must have $T[i] = T[j]$ and $S_{i+1} > S_{j+1}$. But then $r_i > r_j$ by maximality of $i+j$ and the second condition is violated. □

Theorem 7. *The suffix array checker from Figure 6 can be implemented to run using* $\text{sort}(5n)+\text{scan}(2n)$ *I/Os.*

8 Experiments

We have implemented the algorithms in C++ using the g++ 3.2.3 compiler (optimization level `-O2 --omit-framepointer`) and the external memory library STXXL Version 0.52 [9]. Our experimental platform has two 2.0 GHz Intel Xeon processors, one GByte of RAM, and we use four 80 GByte IBM 120GXP disks. Refer to [10] for a performance evaluation of this machine whose cost was 2500 Euro in July 2002. The following instances have been considered:

Table 1: Statistics of the instances used in the experiments.

T	$n = \|T\|$	$\|\Sigma\|$	maxlcp	$\overline{\text{lcp}}$	$\log \overleftrightarrow{\text{lcp}}$
Random2	2^{32}	128	2^{31}	$\approx 2^{29}$	≈ 30
Gutenberg	3 277 099 765	128	4 819 356	45 617	todo
Genome	3 070 128 194	5	21 999 999	454 111	todo
HTML	4 214 295 245	128	102 356	1 108	todo
Source	547 505 710	128	173 317	431	5.80

Random2: Two concatenated copies of a Random string of length $n/2$. This is a difficult instance that is hard to beat using simple heuristics.

Gutenberg: Freely available English texts from `http://promo.net/pg/list.html`.

Genome: The known pieces of the humane genome from `http://genome.ucsc.edu/downloads.html` (status May, 2004). We have normalized this input to ignore the distinction between upper case and lower case letters. The result are characters in an alphabet of size 5 (ACGT and sometime long sequences of "unknown" characters).

HTML: Pages from a web crawl containing only pages from `.gov` domains. These page are filtered so that only text and html code is contained but no pictures and no binary files.

Source: Source code (mostly C++) containing coreutils, gcc, gimp, kde, xfree, emacs, gdb, Linux kernel and Open Office).

We have collected some of these instances at `ftp://www.mpi-sb.mpg.de/pub/outgoing/sanders/`. For a nonsynthetic instance T of length n, our experiments use T itself and its prefixes of the form $T[0, 2^i)$. Table 1 shows statistics of the properties of these instances.

The figure on the next page shows execution time and I/O volume side by side for each of our instance families and for the algorithms nonpipelined doubling, pipelined doubling, pipelined doubling with discarding, pipelined quadrupling, pipelined quadrupling with discarding[5], and DC3. All ten plots share the same x-axis and the same curve labels. Computing all these instances takes about 14 days moving more than 20 TByte of data. Due to these large execution times it was not feasible to run all algorithms for all input sizes and all instances. However, there is enough data to draw some interesting conclusions.

Complicated behavior is observed for "small" inputs up to 2^{26} characters. The main reason is that we made no particular effort to optimize special cases where at least some part of some algorithm could execute internally but STXXL sometime makes such optimizations automatically.

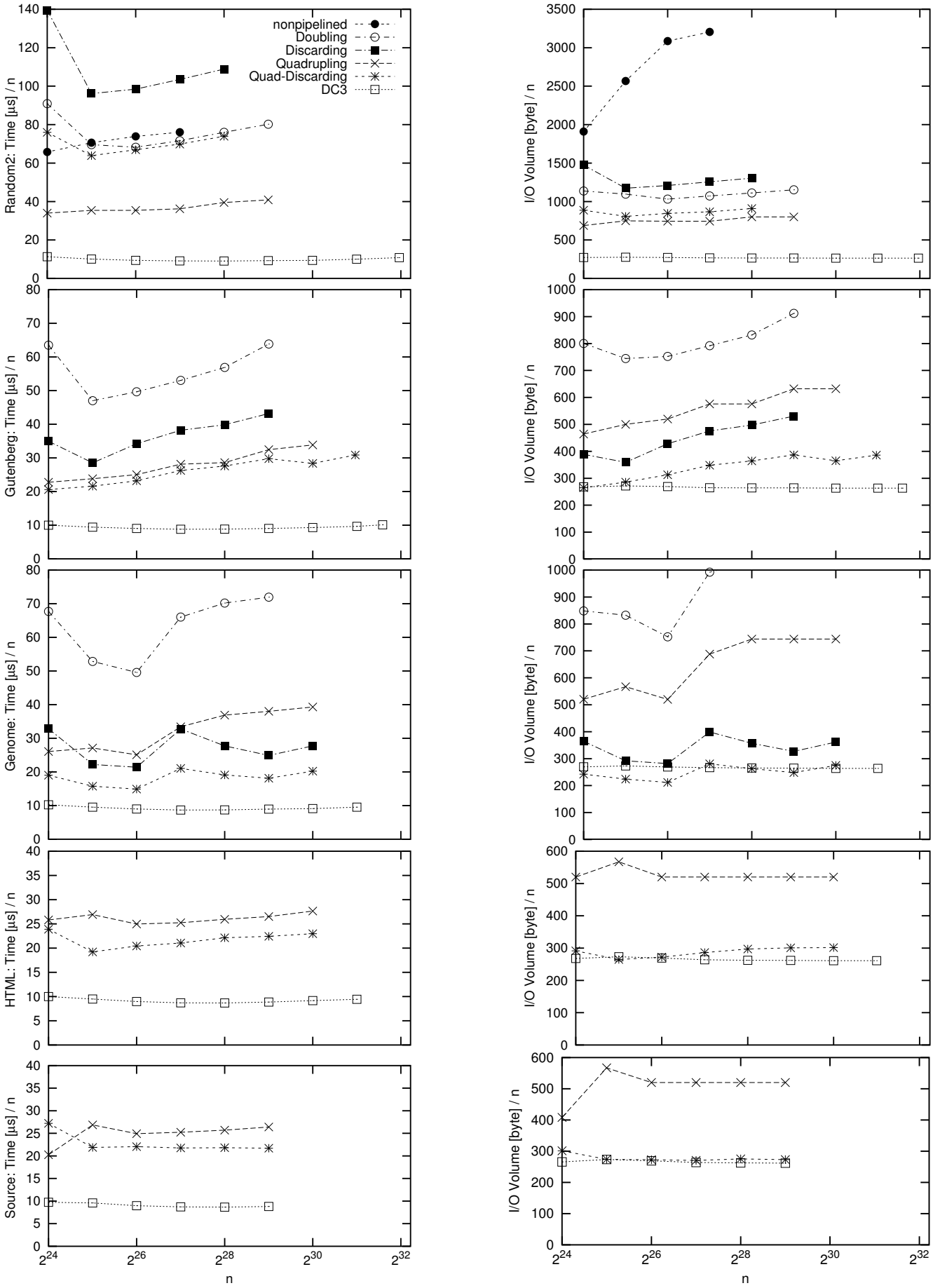
The most important observation is that the DC3-algorithm is always the fastest algorithm and is almost completely insensitive to the input. For all inputs of size more than a GByte, DC3 is at least twice as fast as its closest competitor. With respect to I/O volume, DC3 is sometimes equaled by quadrupling with discarding. This happens for relatively small inputs. Apparently quadrupling has more complex internal work. For example, it compares quadruples during half of its sorting operations whereas DC3 never compares more than triples during sorting. For the difficult synthetic input Random2, quadrupling with discarding is by far outperformed by DC3.

For real world inputs, discarding algorithms turn out to be successful compared to their nondiscarding counterparts. They outperform them both with respect to I/O volume and running time. For random inputs without repetitions the discarding algorithms might actually beat DC3 since one gets inputs with very small values of $\log \overleftrightarrow{\text{lcp}}$.

Quadrupling algorithms consistently outperform doubling algorithms.

Comparing pipelined doubling with nonpipelined doubling in the top pair of plots (instance Random2) one can see that pipelining brings a huge reduction of I/O volume whereas the execution time is affected much less — a clear indication that our algorithms are dominated by internal calculations. We do not show the nonpipelined algorithm for the other inputs since the relative performance compared to pipelined doubling should remain about the same.

[5]The discarding algorithms we have implemented need slightly more I/Os and perhaps more complex calculations than the newer algorithms described in Section 4.


nonpipelined
Doubling
Discarding
Quadrupling
Quad-Discarding
DC3
Random2: Time [µs] / n
Gutenberg: Time [µs] / n
Genome: Time [µs] / n
HTML: Time [µs] / n
Source: Time [µs] / n
I/O Volume [byte] / n
n

A comparison of the new algorithms with previous algorithms is more difficult. The implementation of [7] works only up to 2GByte of total external memory consumption and would thus have to compete with space efficient internal algorithms on our machine. At least we can compare I/O volume per byte of input for the measurements in [7]. Their best scalable algorithm for the largest real world input tested (26 MByte of text from the Reuters news agency) is nonpipelined doubling with a simple form of discarding. This algorithm needs an I/O volume of 1303 Bytes per character of input. The DC3-algorithm about 5 times less I/Os. Furthermore, it is to be expected that the lead gets bigger for larger inputs. The GBS algorithm [12] needs 486 bytes of I/O per character for this input in [7], i.e., even for this small input DC3 already outperforms the GBS algorithm. We can also attempt a speed comparison in terms of clock cycles per byte of input. Here [7] needs 157 000 cycles per byte for doubling with simple discarding and 147 000 cycles per byte for the GBS algorithm whereas DC3 needs only about 20 000 cycles. Again, the advantage should grow for larger inputs in particular when comparing with the GBS algorithm.

The following small table shows the execution time of DC3 for 1 to 8 disks on the 'Source' instance.

D	1	2	4	6	8
$t[mus/\text{byte}]$	13.96	9.88	8.81	8.65	8.52

We see that adding more disks gives only very small speedup. (And we would see very similar speedups for the other algorithms except nonpipelined doubling). Even with 8 disks, DC3 has an I/O rate of less than 30 MByte/s which is less than the peak performance of a *single* disk (45 MByte/s). Hence, by more effective overlapping of I/O and computation it should be possible to sustain the performance of eight disks using a single cheap disk so that even very cheap PCs ($\approx$) could be used for external suffix array construction.

9 Conclusion

Our efficient external version of the DC3-algorithm is theoretically optimal and clearly outperforms all previous algorithms in practice. Since all practical previous algorithms are asymptotically suboptimal and dependent on the inputs, this closes a gap between theory and practice. DC3 even outperforms the pipelined quadrupling-with-discarding algorithm even for real world instances. This underlines the practical usefulness of DC3 since a mere comparison with the relatively simple, nonpipelined previous implementations would have been unfair.

As a side effect, the various generalizations of doubling yield an interesting case study for the systematic design of pipelined external algorithms.

The most important practical question is whether constructing suffix arrays in external memory is now feasible. We believe that the answer is a careful 'yes'. We can now process $4 \cdot 10^9$ characters over night on a low cost machine. Two orders of magnitude more than in [7] in a time faster or comparable to previous internal memory computations [23, 20] on more expensive machines.

There are also many opportunities to scale to even larger inputs. For example, one could exploit that about half of the sorting operations are just permutations which should be implementable with less internal work than general sorting. It should also be possible to better overlap I/O and computation. More interestingly, there are many ways to parallelize. On a small scale, pipelining allows us to run several sorters and one streaming thread in parallel. On a large scale DC3 is also perfectly parallelizable [16]. Since the algorithm is largely compute bound, even cheap switched Gigabit-Ethernet should allow high efficiency (DC3 sorts about 13 MByte/s in our measurements). Considering all these improvements and the continuing advance in technology, there is no reason why it should not be possible to handle inputs that are another two orders of magnitude larger in a few years.

Acknowledgements

We would like to thank Stefan Burkhardt and Knut Reinert for valuable pointers to interesting experimental input. Lutz Kettner helped with the design of STXXL . The html pages were supplied by Sergey Sizov from the information retrieval group at MPII. Christian Klein helped with Unix tricks for assembling the data.

References

[1] M. I. Abouelhoda, S. Kurtz, and E. Ohlebusch. The enhanced suffix array and its applications to genome analysis. In *Proc. 2nd Workshop on Algorithms in Bioinformatics*, volume 2452 of *LNCS*, pages 449–463. Springer, 2002.

[2] A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.

[3] L. Arge, P. Ferragina, R. Grossi, and J. S. Vitter. On sorting strings in external memory. In *29th ACM Symposium on Theory of Computing*, pages 540–548, El Paso, May 1997. ACM Press.

[4] L. Arge, O. Procopiuc, and J. S. Vitter. Implementing I/O-efficient data structures using TPIE. In *10th European Symposium on Algorithms (ESA)*, volume 2461 of *LNCS*, pages 88–100. Springer, 2002.

[5] S. Burkhardt and J. Kärkkäinen. Fast lightweight suffix array construction and checking. In *Proc. 14th Annual Symposium on Combinatorial Pattern Matching*, volume 2676 of *LNCS*, pages 55–69. Springer, 2003.

[6] M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical Report 124, SRC (digital, Palo Alto), May 1994.

[7] A. Crauser and P. Ferragina. Theoretical and experimental study on the construction of suffix arrays in external memory. *Algorithmica*, 32(1):1–35, 2002.

[8] A. Crauser and K. Mehlhorn. LEDA-SM a platform for secondary memory computations. Technical report, MPII, 1998. draft.

[9] R. Dementiev. The stxxl library. documentation and download at `http://www.mpi-sb.mpg.de/~rdementi/stxxl.html`.

[10] R. Dementiev and P. Sanders. Asynchronous parallel disk sorting. In *Proc. 15th Annual Symposium on Parallelism in Algorithms and Architectures*. ACM, 2003. To appear.

[11] M. Farach-Colton, P. Ferragina, and S. Muthukrishnan. On the sorting-complexity of suffix tree construction. *Journal of the ACM*, 47(6):987–1011, 2000.

[12] G. Gonnet, R. Baeza-Yates, and T. Snider. New indices for text: PAT trees and PAT arrays. In W. B. Frakes and R. Baeza-Yates, editors, *Information Retrieval: Data Structures & Algorithms*. Prentice-Hall, 1992.

[13] W.-K. Hon, T.-W. Lam, K. Sadakane, and W.-K. Sung. Constructing compressed suffix arrays with large alphabets. In *Proc. 14th International Symposium on Algorithms and Computation*, volume 2906 of *LNCS*, pages 240–249. Springer, 2003.

[14] W.-K. Hon, K. Sadakane, and W.-K. Sung. Breaking a time-and-space barrier in constructing full-text indices. In *Proc. 44th Annual Symposium on Foundations of Computer Science*, pages 251–260. IEEE, 2003.

[15] J. Kärkkäinen. *Algorithms for Memory Hierarchies*, volume 2625 of *LNCS*, chapter Full-Text Indexes in External Memory, pages 171–192. Springer, 2003.

[16] J. Kärkkäinen and P. Sanders. Simple linear work suffix array construction. In *Proc. 30th International Conference on Automata, Languages and Programming*, volume 2719 of *LNCS*, pages 943–955. Springer, 2003.

[17] T. Kasai, G. Lee, H. Arimura, S. Arikawa, and K. Park. Linear-time longest-common-prefix computation in suffix arrays and its applications. In *Proc. 12thAnnual Symposium on Combinatorial Pattern Matching*, volume 2089 of *LNCS*, pages 181–192. Springer, 2001.

[18] D. K. Kim, J. S. Sim, H. Park, and Kunsoo Park. Linear-time construction of suffix arrays. In *Proc. 14th Annual Symposium on Combinatorial Pattern Matching*. Springer, June 2003. To appear.

[19] P. Ko and S. Aluru. Space efficient linear time construction of suffix arrays. In *Proc. 14th Annual Symposium on Combinatorial Pattern Matching*. Springer, June 2003. To appear.

[20] T.-W. Lam, K. Sadakane, W.-K. Sung, and Siu-Ming Yiu. A space and time efficient algorithm for constructing compressed suffix arrays. In *Proc. 8th Annual International Conference on Computing and Combinatorics*, volume 2387 of *LNCS*, pages 401–410. Springer, 2002.

[21] U. Manber and G. Myers. Suffix arrays: A new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, October 1993.

[22] G. Manzini and P. Ferragina. Engineering a lightweight suffix array construction algorithm. In *Proc. 10th Annual European Symposium on Algorithms*, volume 2461 of *LNCS*, pages 698–710. Springer, 2002.

[23] K. Sadakane and T.Shibuya. Indexing huge genome sequences for solving various problems. *Genome Informatics*, 12:175–183, 2001.

[24] A. Silberschatz, H. F. Korth, and S. Sudarshan. *Database System Concepts*. McGraw-Hill, 4th edition, 2001.

[25] J. S. Vitter and E. A. M. Shriver. Algorithms for parallel memory, I: Two level memories. *Algorithmica*, 12(2/3):110–147, 1994.

A An Introductory Example For Pipelining

To motivate the idea of pipelining let us first analyze the constant factor in a naive implementation of the doubling algorithm from Figure 1. For simplicity assume for now that inputs are not too large so that sorting m words can be done in $4m/DB$ I/Os using two passes over the data. For example, one run formation phase could build sorted runs of size M and one multiway merging phase could merge the runs into a single sorted sequence.

Line (1) sorts n triples and hence needs $12n/DB$ I/Os. Naming in Line (2) scans the triples and writes name-index pairs using $3n/DB + 2n/DB = 5n/DB$ I/Os. The naming procedure can also determine whether all names are unique now, hence the test in Line (3) needs no I/Os. Sorting the pairs in P in Line (4) costs $8n/DB$ I/Os. Scanning the pairs and producing triples in Line (5) costs another $5n/DB$ I/Os. Overall, we get $(12+5+8+5)n/DB = 30n/DB$ I/Os for each iteration.

This can be radically reduced by interpreting the sequences S and P not as files but as pipelines similar to the pipes available in UNIX. In the beginning we explicitly scan the input T and produce triples for S. We do not count these I/Os since they are not needed for the subsequent iterations. The triples are not output directly but immediately fed into the run formation phase of the sorting operation in Line (1). The runs are output to disk ($3n/DB$ I/Os). The multiway merging phase reads the runs ($3n/DB$ I/Os) and directly feeds the sorted triples into the naming procedure called in Line (2) which generates pairs that are immediately fed into the run formation process of the next sorting operation in Line (3) ($2n/DB$ I/Os). The multiway merging phase ($2n/DB$ I/Os) for Line (3) does not write the sorted pairs but in Line (4) it generates triples for S that are fed into the pipeline for the next iteration. We have eliminated all the I/Os for scanning and half of the I/Os for sorting resulting in only $10n/DB$ I/Os per iteration — only one third of the I/Os needed for the naive implementation.

Note that pipelining would have been more complicated in the more traditional formulation where Line (3) sorts P directly by the index i. In that case, a pipelining formulation would require a FIFO of size 2^k to produce a shifted sequences. When $2^k > M$ this FIFO would have to be maintained externally causing $2n/DB$ additional I/Os per iteration, i.e., our modification simplifies the algorithm and saves up to 20 % I/Os.

B Proof of Theorem 2

Proof. The basic observation is that all streaming nodes within an SCC C of G' must be executed together exchanging data through their internal buffers — if any node from C is excluded it will eventually stall the computation because input or output buffer fill up.

Now assume that G fulfills the requirements. We schedule the computations for each SCC of G' in topologically sorted order. First consider an SCC C of streaming nodes. We perform in a single pass all the computations of the streaming nodes in C, reading from the file nodes with edges entering C, writing to the file nodes with edges coming from C, performing the first phase of sorting (e.g., run formation) of the sorting nodes with edges coming from C, and performing the last phase of sorting (e.g. multiway merging) for the sorting nodes with edges entering C. The requirement on the buffer sizes ensures that there is sufficient internal memory. The topological sorting ensures that all the data from incoming edges is available. Since there are only streaming nodes in C, data can freely flow through them respecting the topological sorting of G.[6]

[6] In our implementations the detailed scheduling within the components is done by the user to keep the overhead small. However, one could also schedule them automatically, possibly using multithreading.

When a sorting node is encountered as an SCC we may have to perform I/Os to make sure that the final phase can incrementally produce the sorted elements. However for a sorting volume of $\mathcal{O}(M^2/B)$, multiway merging only needs the run formation phase that will already be done and the final merging phase that will be done later. For SCCs consisting of file nodes we do nothing.

Now assume the G violates the requirements. If there is an SCC that exceeds its buffer requirements, there is no systematic way to execute all its nodes together.

If an SCC C of G' contains a sorting node v, there must be a streaming node w that directly or indirectly needs input from v, i.e., it cannot start executing before v starts to produce output. Node v cannot produce any output before it did not see its complete input. This input directly or indirectly depends on some other streaming node u in C. Since u and w are in the same SCC, they have to be executed together. But the data dependencies above make this impossible. The argument for a file node within an SCC is analogous. □