# Better External Memory Suffix Array Construction

Roman Dementiev
Fakultät für Informatik, Universität Karlsruhe, Germany
and
Juha Kärkkäinen
Department of Computer Science, University of Helsinki, Finland
and
Jens Mehnert
Max-Planck-Institut für Informatik, Saarbrücken, Germany
and
Peter Sanders
Fakultät für Informatik, Universität Karlsruhe, Germany

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 has 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 4GBytes in hours on a low cost machine.

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

Categories and Subject Descriptors: E.1 [**Data Structures**]: suffix arrays; F.2.2 [**Nonnumerical Algorithms and Problems**]: algorithms for strings; D.4.2 [**Storage Management**]: secondary storage

General Terms: Algorithms, Experimentation, Performance

Additional Key Words and Phrases: algorithm engineering, algorithms for strings, external memory, I/O-efficient, large data sets, secondary memory, suffix array

## 1. INTRODUCTION

The suffix array [Manber and Myers 1993; Gonnet et al. 1992], a lexicographically sorted array of the suffixes of a string, has numerous applications, e.g., in string matching [Manber and Myers 1993; Gonnet et al. 1992], genome analysis [Abouelhoda et al. 2002] and text compression [Burrows and Wheeler 1994]. For example, one can use it as full text index: To find all occurrences of a pattern $P$ in a text $T$

Authors addresses: Roman Dementiev, Peter Sanders Fakultät für Informatik, Universität Karlsruhe, 76128 Karlsruhe, Germany, email: [dementiev,sanders]@ira.uka.de; Juha Kärkkäinen Department of Computer Science, FIN-00014 University of Helsinki, Finland, email: Juha.Karkkainen@cs.helsinki.fi; Jens Mehnert Max-Planck-Institut für Informatik, Stuhlsatzenhausweg 85, 66123 Saarbrücken, Germany, email: jmehnert@mpi-sb.mpg.de.

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 [Kärkkäinen et al. 2006; Kim et al. 2003; Ko and Aluru 2003]. One of the linear time algorithms [Kärkkäinen et al. 2006] is very simple and can also be adapted to obtain an optimal algorithm for external memory: The DC3-algorithm [Kärkkäinen et al. 2006] constructs a suffix array of a text $T$ of length $n$ using $\mathcal{O}(\text{sort}(n))$ I/Os where $\text{sort}(n)$ is the number of I/Os needed for sorting the characters of $T$.

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 [Crauser and Ferragina 2002] 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.

### 1.1 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.

Let \$ be a special character that 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 $T_{\text{SA}[i]} < T_{\text{SA}[j]}$ whenever $0 \leq i < j < n$. Let $\text{lcp}(i, j)$ denote the longest common prefix length of $\text{SA}[i]$ and $\text{SA}[j]$ ($\text{lcp}(i, j) = 0$ if $i < 0$ or $j \geq n$). Then $\text{dps}(i) := 1 + \max\{\text{lcp}(i-1, i), \text{lcp}(i, i+1)\}$ is the distinguishing prefix size of $T_i$. 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.

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

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

$$\log \text{dps} := \frac{1}{n} \sum_{0 \le i < n} \log(\text{dps}(i)) \tag{3}$$

The I/O model [Vitter and Shriver 1994] 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 [Vitter and Shriver 1994]. 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 $\text{scan}(x) = \lceil x/(DB) \rceil$ for sequentially reading or writing $x$ words and $\text{sort}(x) \approx \frac{2x}{DB} \left\lceil \log_{M/B} \frac{x}{M} \right\rceil$ for sorting $x$ words of data (not counting the $2\text{scan}(x)$ 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 $\langle i : i \text{ is prime} \rangle = \langle 2, 3, 5, 7, 11, 13, \ldots \rangle$ in that order.

### 1.2 Overview.

In Section 2 we present the *doubling algorithm* [Arge et al. 1997; Crauser and Ferragina 2002] for suffix array construction that has I/O complexity $\mathcal{O}(\text{sort}(n \log \text{maxlcp}))$. 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 systematically introduces the technique of *pipelined* processing of sequences which saves a factor of at least two in I/Os for many external algorithms and is supported by our external memory library STXXL [Dementiev et al. 2005]. 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 when their position in the suffix array is already known. This leads to an algorithm with I/O complexity $\mathcal{O}(\text{sort}(n \log \text{dps}))$ improving on a previous discarding algorithm with I/O complexity $\mathcal{O}(\text{sort}(n \log \text{dps}) + \text{scan}(n \log \text{maxlcp}))$ [Crauser and Ferragina 2002]. 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 $\mathcal{O}(\text{sort}(n))$ in Section 6 and its generalization in Section 7 conclude 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 8.

In Section 9 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 and a faster high-end system. The theoretically optimal algorithm turns out to be the winner closely followed by quadrupling with discarding.

Section 10 summarizes the overall results and discusses how even larger suffix arrays could be build.

### 1.3 More Related Work.

The first I/O optimal algorithm for suffix array construction [Farach-Colton et al. 2000] is based on suffix tree construction and introduced the basic divide-and-conquer approach that is also used by DC3. However, the algorithm from [Farach-Colton et al. 2000] is very complex such that the constant factors hidden in the $\mathcal{O}()$-notation are very high. Therefore implementing the algorithm looks not promising.

There is an extensive implementation study for external suffix array construction by Crauser and Ferragina [Crauser and Ferragina 2002]. They implement several nonpipelined variants of the doubling algorithm [Arge et al. 1997] including one that discards unique suffixes. However, this variant of discarding still needs to scan all unique tuples in each iteration. Our discarding algorithm eliminates these scanning costs which dominate the I/O volume for many inputs. Interestingly, an algorithm that fares very well in the study of [Crauser and Ferragina 2002] is the GBS-algorithm [Gonnet et al. 1992] that takes $\mathcal{O}\big(\frac{N}{M}\text{scan}(n)\big)$ I/Os. 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. Moreover, the GBS-algorithm is quite expensive with respect to internal work, which contributes significantly to the running time on our system as shown by the experiments. 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 [Manzini and Ferragina 2002; Burkhardt and Kärkkäinen 2003] and even more compact full-text indexes [Lam et al. 2002; Hon et al. 2003; Hon et al. 2003]. 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 [Sadakane and T.Shibuya 2001; Lam et al. 2002]. One [Lam et al. 2002] computes the compressed suffix array on a PC with 3 GBytes of memory in 21 hours. 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 [Sadakane and T.Shibuya 2001] uses a supercomputer with 64 GBytes of memory and needs 7 hours. Our algorithms have comparable speed using external memory.

Suffix arrays are not used in search engines, instead they use simpler data structures like inverted word indexes. These data structures are fast for short simple

```
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), (c, i) = P[j], (c', i + 2^k) = P[j + 1]⟩   (5)

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
```

Fig. 1. The doubling algorithm.

queries, however, if the query is a long phrase they cannot guarantee the optimal query time achievable with suffix arrays.

Pipelining to reduce I/Os is well known technique in executing database queries [Silberschatz et al. 2001]. However, previous algorithm libraries for external memory [Arge et al. 2002; Crauser and Mehlhorn 1998] do not support it. We decided quite early in the design of our library STXXL [Dementiev et al. 2005] 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. DOUBLING ALGORITHM

Figure 1 gives pseudocode for the doubling algorithm [Arge et al. 1997; Crauser and Ferragina 2002]. 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 variants of this algorithm, our formulation never actually builds the resulting string of names. Rather, it manipulates a sequence $P$ of pairs $(c, i)$ where each name $c$ 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 the pair $(i \bmod 2^k, i \operatorname{div} 2^k)$. [1] 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 optimization described in Section 3.

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

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 2.1. *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. For simplicity assume for now that inputs are not too large ($\mathcal{O}\big(M^2/B\big)$) 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 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.

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 $b(s)$.[2] 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[3]. Edges are labeled with the number of machine words $w(e)$ flowing between two nodes. In the proof of Theorem 3.2 the flow graph for the pipelined doubling algorithm is shown. We will see somewhat more complicated graphs in Sections 4 and 6. The following theorem 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 3.1. *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*

$$\sum_{e \in E \cap (F \times V \cup V \times F)} \text{scan}(w(e)) + \sum_{e \in E \cap (V \times R)} \text{sort}(w(e)) \tag{4}$$

*I/Os iff the following conditions are fulfilled: 1. $G$ is a DAG. 2. Consider the undirected graph $G' = (S, \{\{u, v\} : (u, v) \in E \cap (S \times S)\})$ induced by the streaming nodes. The total buffer requirement of each connected component $C$ of $G'$ plus the buffer requirements of the nodes directly connected to $C$ in $G$ do not exceed the internal memory size $M$.*

PROOF. The basic observation is that all streaming nodes within a connected component $C$ 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 an input or an output buffer fills up. [4]

Now assume that $G$ fulfills the requirements. We schedule the computations for each connected component (CC) of $G'$ in topologically sorted order. First consider a CC $C$ 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 $G$.[5]

[2]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.

[3]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.

[4]For example, in Figure 7, nodes 13 and 14 are executed together. Any set of pipelining node not separated from each other by file nodes or sorting nodes has data dependencies that require them to be executed together.

[5]In our implementations the detailed scheduling within the components is done by the user to

When a sorting node is encountered as a CC 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 CCs consisting of file nodes we do nothing. □

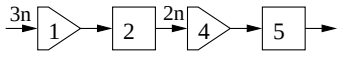
Theorem 3.1 can be used to design and analyze pipelined external memory algorithms in a systematic way. All we have to do is to give a data flow graph that fulfills the requirements and we can then read off the I/O complexity. Using the relations $a \cdot \mathrm{scan}(x) = \mathrm{scan}(a \cdot x) + \mathcal{O}(1)$ and $a \cdot \mathrm{sort}(x) \leq \mathrm{sort}(a \cdot x) + \mathcal{O}(1)$, we can represent the result in the form $\mathrm{scan}(x) + \mathrm{sort}(y) + \mathcal{O}(1)$, 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 algorithm (e.g., $\approx 2x/DB$ for $x \ll M^2/DB$ and $M \gg DB$) but this may not be desirable because we lose information. In particular, scanning implies less internal work and can usually be implemented using *bulk I/Os* in the sense of [Crauser and Ferragina 2002] (we then need larger buffers $b(v)$ for file nodes) whereas sorting requires many random accesses for information theoretic reasons [Aggarwal and Vitter 1988].

We will also draw data flow graphs with cycles. These will be abbreviations for larger flow graphs with all cycles unrolled a number of times clear from the context.

Now we apply Theorem 3.1 to the doubling algorithm:

THEOREM 3.2. *The doubling algorithm from Figure 1 can be implemented to run using* $\mathrm{sort}(5n)\,\lceil \log(1 + \mathrm{maxlcp}) \rceil + \mathcal{O}(\mathrm{sort}(n))$ *I/Os.*

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




□ streaming node ▷ sorting node

After $\lceil \log(1 + \mathrm{maxlcp}) \rceil$ iterations, the algorithm finishes. The $\mathcal{O}(\mathrm{sort}(n))$ 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 [Dementiev et al. 2005]. 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

keep the overhead small. However, one could also schedule them automatically, possibly using multithreading.

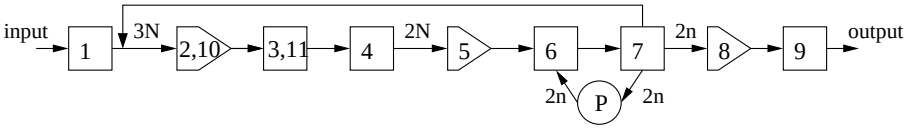



Fig. 2. Data flow graph for the *doubling + discarding*. The numbers refer to line numbers in Figure 4. The edge weights are sums over the whole execution with $N = n \log \mathrm{dps}$.

```
Function name2(S : Sequence of Pair)
    q:= q′:= 0;   (ℓ, ℓ′):= ($, $)
    result:= ⟨⟩
    foreach ((c, c′), i) ∈ S do
        if c ≠ ℓ then q:= q′:= 0; (ℓ, ℓ′):= (c, c′)
        else if c′ ≠ ℓ′ then q′:= q;   ℓ′:= c′
        append (c + q′, i) to result
        q++
    return result
```

Fig. 3. The alternative naming procedure.

`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. The sorting implementations run in optimal $\mathcal{O}\left(\frac{x}{DB}\log_{M/B}\frac{x}{M}\right)$ I/Os, scanning and file node implementations in $\mathcal{O}\left(\frac{x}{DB}\right)$ I/Os, taking the advantage of parallel disks.

## 4. DISCARDING

Let $c_i^k$ be the lexicographic name of $T[i, i + 2^k)$, i.e., the value paired with $i$ at iteration $k$ in Figure 1. Since $c_i^k$ is the number of strictly smaller substrings of length $2^k$, it is a non-decreasing function of $k$. More precisely, $c_i^{k+1} - c_i^k$ is the number of positions $j$ such that $c_j^k = c_i^k$ but $c_{j+2^k}^k < c_{i+2^k}^k$. This provides an alternative way of computing the names given in Figure 3.

Another consequence of the above observation is that if $c_i^k$ is unique, i.e., $c_j^k \neq c_i^k$ for all $j \neq i$, then $c_i^h = c_i^k$ for all $h > k$. The idea of the discarding algorithm is to take advantage of this, i.e., discard pair $(c, i)$ from further iterations once $c$ is unique. A key to this is the new naming procedure in Figure 3, because it works correctly even if we exclude from $S$ all tuples $((c, c'), i)$, where $c$ is unique. Note, however, that we cannot exclude $((c, c'), i)$ if $c'$ is unique but $c$ is not. Therefore, we will *partially* discard $(c, i)$ when $c$ is unique. We will *fully* discard $(c, i) = (c_i^k, i)$ when also either $c_{i-2^k}^k$ or $c_{i-2^{k+1}}^k$ is unique, because then in any iteration $h > k$,

```
Function doubling + discarding(T)
  S:= ⟨((T[i], T[i + 1]), i) : i ∈ [0, n)⟩                          (1)
  sort S                                                            (2)
  U:= name(S)     // undiscarded                                    (3)
  P:= ⟨⟩          // partially discarded
  F:= ⟨⟩          // fully discarded
  for k := 1 to ⌈log n⌉ do
    mark unique names in U                                          (4)
    sort U by (i mod 2^k, i div 2^k)                                (5)
    merge P into U; P:= ⟨⟩                                          (6)
    S:= ⟨⟩; count:= 0
    foreach (c, i) ∈ U do                                           (7)
      if c is unique then
        if count < 2 then append (c, i) to F
      else append (c, i) to P
      count:= 0
    else
      let (c', i') be the next pair in U
      append ((c, c'), i) to S
      count++
  if S = ∅ then
    sort F by first component                                       (8)
    return ⟨i : (c, i) ∈ F⟩                                         (9)
  sort S                                                            (10)
  U:= name2(S)                                                      (11)
```

Fig. 4. The doubling with discarding algorithm.

the first component of the tuple $((c^h_{i-2^h}, c^h_i), i - 2^h)$ must be unique. The final algorithm is given in Figure 4.

THEOREM 4.1. *Doubling with discarding can be implemented to run using* $\mathrm{sort}(5n \log \mathrm{dps}) + \mathcal{O}(\mathrm{sort}(n))$ *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 at most $N = n \log \mathrm{dps}$ tuples are processed in each sorting step over the whole execution 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 $\lceil \log(1 + \mathrm{dps}(i)) \rceil$ 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 $n$. □

A slightly different algorithm with the same asymptotic complexity is described in [Kärkkäinen 2003]. The discarding algorithm in [Crauser and Ferragina 1999; 2002] does partial but not full discarding, adding the term $\mathcal{O}(\mathrm{scan}(n \log \mathrm{maxlcp}))$ to its complexity.

**Function** *atupling*$(T)$
  $S := \langle((T[i], T[i+1], \ldots, T[i+a-1]), i) : i \in [0, n)\rangle$
  **for** $k := 1$ **to** $\lceil \log_a n \rceil$ **do**
    sort $S$
    $P := name(S)$
    **invariant** $\forall (c, i) \in P$ :$c$ is a lexicographic name for $T[i, i + a^k)$
    **if** the names in $P$ are unique **then return** $\langle i : (c, i) \in P\rangle$
    sort $P$ by ($i$ **mod** $a^k, i$ **div** $a^k$))
    $S := \langle((c_0, \ldots, c_q, \ldots, c_{a-1}), i) : j \in [0, n),\ (c_q, i + q \cdot a^k) = P[j + q], q \in [0, a)\rangle$

Fig. 5. The $a$-tupling algorithm.

## 5. FROM DOUBLING TO $A$-TUPLING

It is straightforward to generalize the doubling algorithms from Figures 1 and 4 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. The pseudocode of the generalized doubling algorithm without discarding is presented in Figure 5.

THEOREM 5.1. *The $a$-tupling algorithm can be implemented to run using*

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

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

*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 $a = 5$ (Table I). 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.

Table I. I/O requirements for different variants of the $a$-tupling algorithm. The entries specify the variable $x$ defined in the column headings. $+\mathcal{O}(\text{sort}(n))$ terms are omitted.

| $a$ | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|
| $(a+3)/\log a$ | 5.00 | 3.78 | 3.50 | **3.45** | 3.48 | 3.56 |

## 6. A PIPELINED I/O-OPTIMAL ALGORITHM

The following three-step algorithm outlines a linear time algorithm for suffix array construction [Kärkkäinen et al. 2006]:

(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 6 gives pseudocode for an external implementation of this algorithm and Figure 7 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 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 3.1 that edges between streaming nodes must not cross sorting nodes. This problem 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 6.1. *The DC3 algorithm from Figure 6 can be implemented to run using* $\mathrm{sort}(30n) + \mathrm{scan}(6n)$ *I/Os.*

PROOF. Let $V(n)$ denote the number of I/Os for the external DC3 algorithm.

**Function** *DC3*$(T)$

| | | |
|---|---|---|
| $S$:= $\langle((T[i], i+2]), i) : i \in [0, n), i \bmod 3 \neq 0\rangle$ | // mod12 suffixes | (1) |
| sort $S$ by the first component | // sort triples | (2) |
| $P$:= $name(S)$ | // name triples | (3) |
| **if** the names in $P$ are not unique **then** | | |
| sort the $(i, r) \in P$ by $(i \bmod 3, i \operatorname{div} 3)$ | // build recursive input | (4) |
| $\mathrm{SA}^{12}$:= $DC3(\langle c : (c, i) \in P\rangle)$ | // recurse | (5) |
| $P$:= $\langle(j + 1, \mathrm{SA}^{12}[j]) : j \in [0, 2n/3)\rangle$ | | (6) |
| sort $P$ by the second component | // inverse SA of sample | (7) |
| $S_0$:= $\langle(T[i], T[i+1], c', c'', i) : i \bmod 3 = 0, (c', i+1), (c'', i+2) \in P\rangle$ | | (8) |
| $S_1$:= $\langle(c, T[i], c', i) : i \bmod 3 = 1, (c, i), (c', i+1) \in P\rangle$ | | (9) |
| $S_2$:= $\langle(c, T[i], T[i+1], c'', i) : i \bmod 3 = 2, (c, i), (c'', i+2) \in P\rangle$ | | (10) |
| sort $S_0$ by components 1,3 | // sort mod0 suffixes | (11) |
| sort $S_1$ and $S_2$ by component 1 | // resort mod12 suffixes | (12) |
| $S$:= $merge(S_0, S_1, S_2)$ using 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** $\langle$last component of $s : s \in S\rangle$ | | (14) |

Fig. 6. The DC3-algorithm.

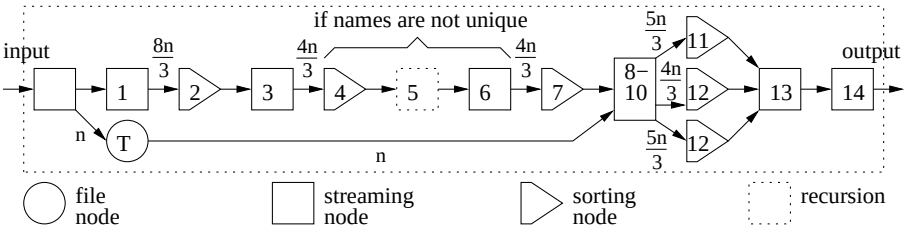



Fig. 7. Data flow graphs for the DC3 algorithm. The numbers refer to line numbers in Figure 6.

Using Theorem 3.1 and the data flow diagram from Figure 7 we can conclude that

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

This recurrence has the solution $V(n) \leq 3(\mathrm{sort}(10n) + \mathrm{scan}(2n)) \leq \mathrm{sort}(30n) + \mathrm{scan}(6n)$. 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. GENERALIZED DIFFERENCE COVER ALGORITHM

DC3 computes the suffix array of two thirds of the suffixes in its recursion. In the generalized algorithm DCX [Kärkkäinen et al. 2006] one tries to reduce the number of sample suffixes, which might decrease the cost of the recursion.

The algorithm DCX chooses the sample of suffixes starting at indexes $I_X = \{i \mid i \mod X \in C_X\}$ (for DC3 $X = 3$ and $C_3 = \{1, 2\}$). For any given $X$ the set $C_X$

Table II. Minimum difference covers.

| $X$ | $C'_X$ |
|---|---|
| 3 | $\{0, 1\}$ |
| 7 | $\{0, 1, 3\}$ |
| 13 | $\{0, 1, 3, 9\}$ |
| 21 | $\{0, 1, 6, 8, 18\}$ |
| 31 | $\{0, 1, 3, 8, 12, 18\}$ |
| 39 | $\{0, 1, 16, 20, 22, 27, 30\}$ |
| 57 | $\{0, 1, 9, 11, 14, 35, 39, 51\}$ |
| 73 | $\{0, 1, 3, 7, 15, 31, 36, 54, 63\}$ |
| 91 | $\{0, 1, 7, 16, 27, 56, 60, 68, 70, 73\}$ |
| 95 | $\{0, 1, 5, 8, 18, 20, 29, 31, 45, 61, 67\}$ |
| 133 | $\{0, 1, 32, 42, 44, 48, 51, 59, 72, 77, 97, 111\}$ |

must be chosen such that $|C_X|$ is minimal and the order of the remaining suffixes can be reconstructed using the sample suffixes. To fulfil these requirements one uses the *minimum difference covers* [Haanpää 2004] of $\mathbb{Z}_X$ ($\mathbb{Z}_X$ is the integers modulo $X$). For a subset $C'$ of a finite Abelian group $G$, we define $d(C') = \{a - b \,|\, a, b \in C'\}$. If $d(C') = G$, we call $C'$ a difference cover of $G$. [Haanpää 2004] contains minimum difference covers $C'_X$ of $\mathbb{Z}_X$ for primes $X$ up to 133 (see also Table II). The algorithm DCX sets $C_X = \{j \,|\, X - j - 1 \in C'_X\}$.

Now we find the number of I/Os needed by a recursion of the DCX algorithm: sorting $S$ by $T[i, i + X - 1]$ (Line (2) in Figure 6) costs $\mathrm{sort}((X + 1)n \cdot \frac{|C_X|}{X})$ I/Os, writing and reading $T$ takes $\mathrm{scan}(2n)$ I/Os, building the input for the recursion (Line (4)) needs $\mathrm{sort}(2n \cdot \frac{|C_X|}{X})$ I/Os, permuting in Line (7) incurs $\mathrm{sort}(2n \cdot \frac{|C_X|}{X})$ I/Os, sorting the merge tuples (Lines (11)–(12)) needs $\mathrm{sort}(\delta_X n)$ I/Os, where $\delta_X$ is the average merge tuple size ($\delta_3 = \frac{5+4+5}{3}$). Let $V_X(n)$ be the number of I/Os for the DCX algorithm.

$$V_X(n) \leq \mathrm{sort}(((X + 5)\tfrac{|C_X|}{X} + \delta_X)n) + \mathrm{scan}(2n) + V_X(\tfrac{|C_X|}{X}n)$$

This recurrence has the solution

$$V_X(n) \leq \mathrm{sort}(n\frac{(X + 5)|C_X| + X\delta_X}{X - |C_X|}) + \mathrm{scan}(2n\frac{X}{X - |C_X|})$$

To analyse $V_X(n)$ one need to know the values of $\delta_X$ for given $X$. Unfortunately, a simple formula does not exist. Instead, we compute upper bounds for $\delta_X$ using a simple algorithm. Let

$$d_{max}(i) = \max\{k \,|\, i + k \mod X \in C_X \wedge k < X\}$$

be the maximal distance from starting position $i$ to the next sample to the right, i.e. the maximum number of characters needed in a merge tuple. Then the merge tuple size for positions $j$ such that $i \equiv j \mod X$ is $d_{max}(i) + 1 + |C_X|$, because one might need the ranks of all the $|C_X|$ samples to compare two arbitrary merge tuples and one component takes the index value. Hence the average merge tuple size is:

$$\delta_X = 1 + |C_X| + \frac{1}{X}\sum_{0 \leq i < X} d_{max}(i)$$

Table III. I/O volume of DCX. We assume that $\text{sort}(x) = 2 \cdot \text{scan}(x)$ which is realistic using a pipelined sorter with a proper choice of $B$. The total I/O volume is then computed in terms of the scanned I/O volume.

| $X$ | 3 | 7 | 13 | 21 | 31 | 39 | 57 |
|---|---|---|---|---|---|---|---|
| $\lvert C_X \rvert$ | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| sort$[N]$ | 30 | 24.75 | 30.11 | 38.56 | 50.12 | 60.65 | 79.02 |
| scan$[N]$ | 6 | 3.50 | 2.89 | 2.63 | 2.48 | 2.39 | 2.33 |
| Total | 66 | **53** | 63.11 | 79.75 | 102.72 | 123.75 | 160.37 |

Table IV. I/O volume of DCX with the small alphabet optimization.

| $X$ | 3 | 7 | 13 | 21 | 31 | 39 | 57 |
|---|---|---|---|---|---|---|---|
| $\lvert C_X \rvert$ | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| sort$[N]$ | 24.50 | 18.17 | 15.46 | 15.23 | 15.14 | 16.57 | 17.43 |
| scan$[N]$ | 2.46 | 1.63 | 1.20 | 0.96 | 0.80 | 0.75 | 0.61 |
| Total | 51.49 | 37.99 | 32.13 | 31.41 | **31.09** | 33.89 | 35.49 |

Table III presents the computed I/O volume for DCX algorithm with $X \in \{3, 7, 13, 21, 31, 39, 57\}$. The algorithm with the smallest I/O volume is DC7. Recently, it has been experimentally confirmed that DC7 is faster than DC3 [Weese 2006].

Each tuple component of the DCX algorithm is represented as a 32-bit word, which is wasteful for small alphabets. For the genome data with a four character alphabet one can put up to 16 characters needed for a naming tuple in one word. The merge tuple can be compressed similarly. Table IV shows the computed I/O volume of the DCX algorithm that uses this bit optimization in its first recursion and calls DC7 in the further recursions.

## 8. 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 8 and is based on the following result.

LEMMA 8.1 [BURKHARDT AND KÄRKKÄINEN 2003]. *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) $\forall i, j : r_i \leq r_j \Leftrightarrow (T[i], r_{i+1}) \leq (T[j], r_{j+1})$ where $r_i$ denotes the rank of the suffix $S_i$ according to the suffix array.*

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 8.2. *The suffix array checker from Figure 8 can be implemented to run using* $\text{sort}(5n) + \text{scan}(2n)$ *I/Os.*

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

Fig. 8. The suffix array checker.

Table V. Input instances.

| Name | Description |
|---|---|
| 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 human 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 sometimes long sequences of "unknown" characters). |
| HTML | Pages from a web crawl containing only pages from `.gov` domains. These pages 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). |

## 9. EXPERIMENTS

We have implemented the algorithms (except DCX) in C++ using the g++ 3.2.3 compiler (optimization level `-O2 -fomit-frame-pointer`)[6] and the external memory library STXXL Version 0.52 [Dementiev ; Dementiev et al. 2005]. We have run the experiments on two platforms. The first system has two 2.0 GHz Intel Xeon processors (our implementations use only one processor), one GByte of RAM and eight 80 GByte ATA IBM 120GXP disks. Refer to [Dementiev and Sanders 2003] for a performance evaluation of this machine whose cost was 2500 Euro in July 2002. The second platform is a more expensive SMP system with four 64-bit AMD Opteron 1.8 GHz processors, 8 GByte of RAM (we use only one GByte) and eight 73 GByte SCSI Seagate 15000 RPM ST373453LC disks. In our experiments we used four disks if not otherwise specified.

Table V shows the considered input instances. We have collected some of these instances at `http://algo2.iti.uka.de/dementiev/esuffix/instances.shtml` and `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 VI and Figure 9 show statistics of the properties of these instances.

[6]The sources are available under `http://algo2.iti.uka.de/dementiev/esuffix/docu/index.html`.

Table VI. Statistics of the instances used in the experiments.

| $T$ | $n = \|T\|$ | $\|\Sigma\|$ | maxlcp | $\overline{\text{lcp}}$ | log dps |
|---|---|---|---|---|---|
| Random2 | $2^{32}$ | 128 | $2^{31}$ | $\approx 2^{29}$ | $\approx 29.56$ |
| Gutenberg | 3 277 099 765 | 128 | 4 819 356 | 45 617 | 10.34 |
| Genome | 3 070 128 194 | 5 | 21 999 999 | 454 111 | 6.53 |
| HTML | 4 214 295 245 | 128 | 102 356 | 1 108 | 6.99 |
| Source | 547 505 710 | 128 | 173 317 | 431 | 5.80 |

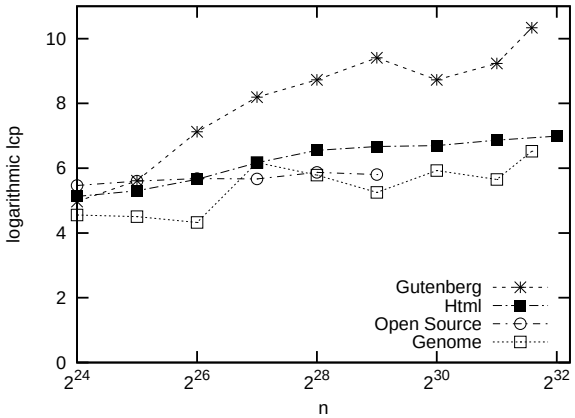



Fig. 9. Statistics of the instances used in the experiments.

Figure 10 shows the execution time and the 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[7], and DC3 running on the Xeon machine. 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. Sometimes STXXL makes such optimizations, e.g. automatically sorting small inputs in internal memory. Another factor is the constant start-up overhead of `stxxl::vector`s which amortizes only with larger inputs. The data size granularity with which `stxxl::vector` loads and stores blocks from/to external memory was not optimized for small inputs.

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 a size of more than a GByte, DC3 is at least twice as fast as its closest competitor.

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

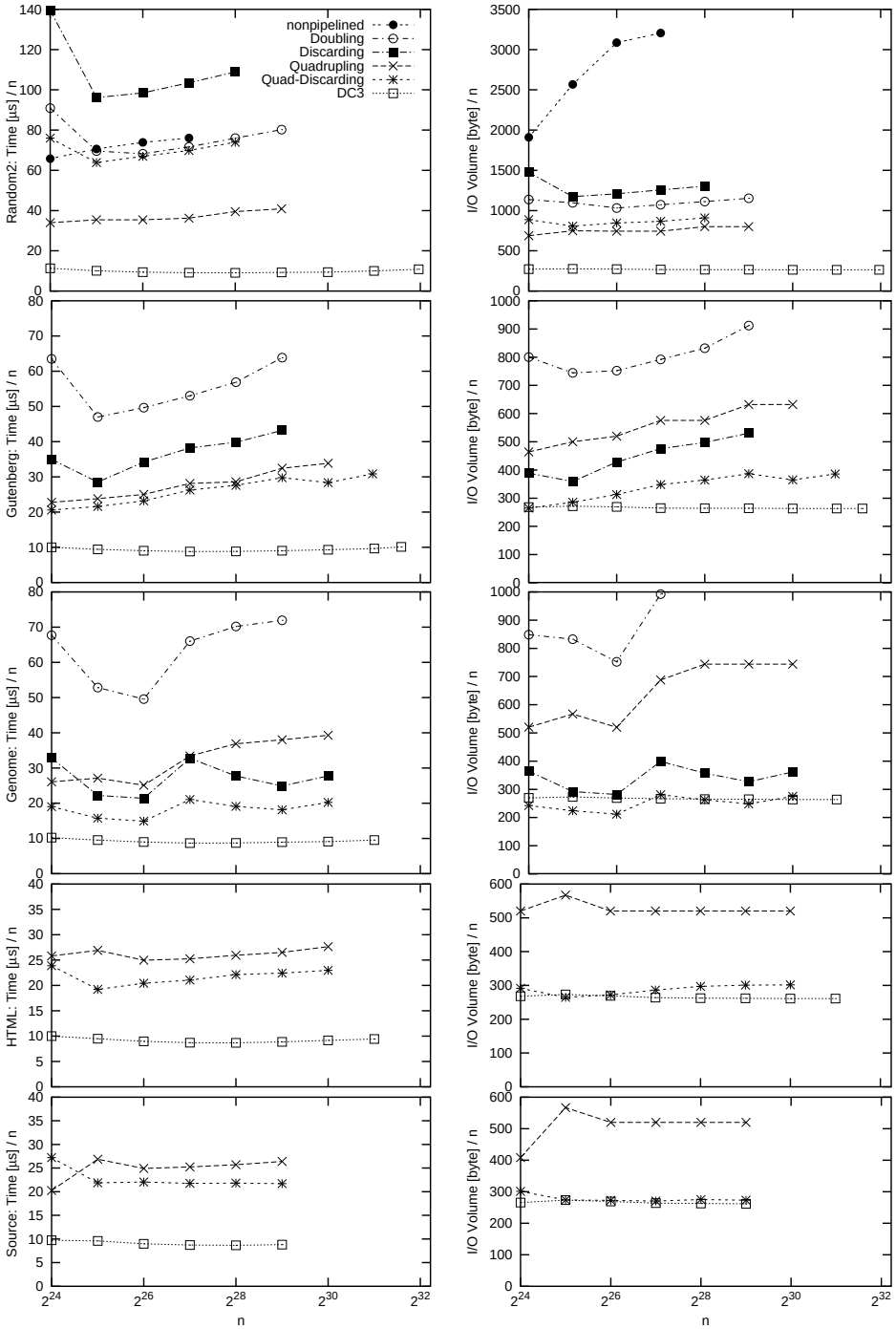



Fig. 10. Execution time (left) and I/O volume (right) for Random2, Gutenberg, Genome, HTML (Xeon machine).

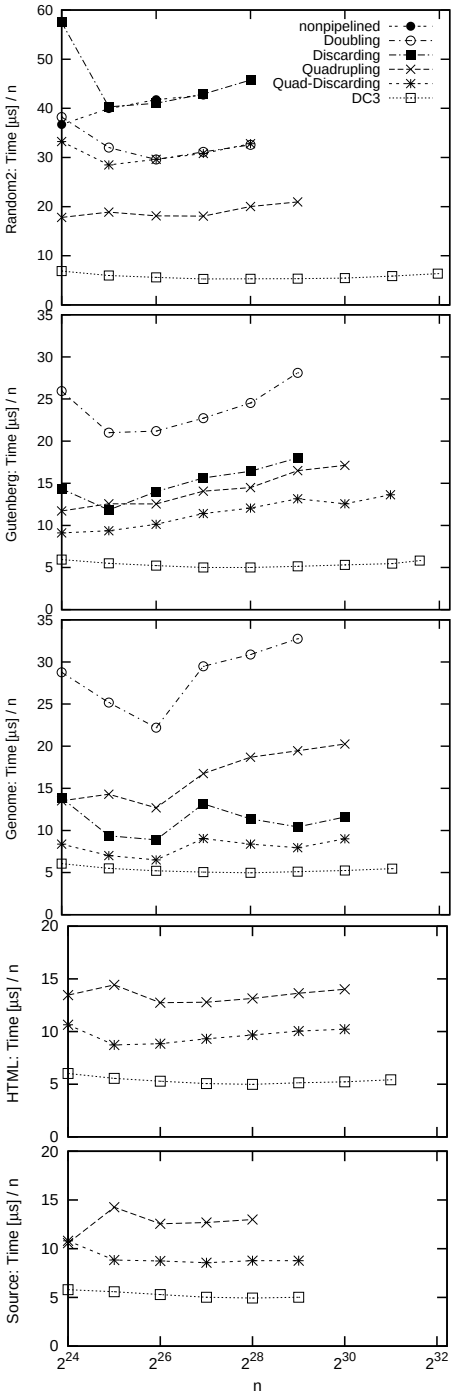



Fig. 11. Execution time for Random2, Gutenberg, Genome, HTML (Opteron machine).

With respect to the I/O volume, DC3 is sometimes equaled by quadrupling with discarding. This happens for relatively small inputs. Apparently quadrupling has more complex internal work.[8] For example, it compares quadruples during half of its sorting operations whereas DC3 compares triples or pairs during sorting. For the difficult synthetic input Random2, quadrupling with discarding is by far outperformed by DC3. Even plain quadrupling, is much faster than quadrupling with discarding. This indicates that the internal logics for discarding is a bottleneck.

For real world inputs, discarding algorithms turn out to be successful compared to their nondiscarding counterparts. They outperform them both with respect to the I/O volume and the running time. This could be explained by the smaller $\log \mathrm{dps}$ values according to Table VI. For random inputs without repetitions the discarding algorithms might actually beat DC3 since one gets inputs with very small values of $\log \mathrm{dps}$.

Quadrupling algorithms consistently outperform doubling algorithms as predicted by the analysis of the I/O complexity in Section 5.

Comparing pipelined doubling with nonpipelined doubling in the top pair of plots (instance Random2) one can see that pipelining brings a huge reduction of the I/O volume whereas the execution time is affected much less — a clear indication that our algorithms are dominated by internal calculations. However, in a setting with a slower I/O subsystem, e.g. a system with a single disk, pipelining gives a significant speedup. We have also made experiments with $D = 1$ which show that pipelined doubling is faster than its nonpipelined version by a factor of 1.9–2.4. We also have reasons to believe that our nonpipelined sorter is more highly tuned than the pipelined one so that the advantage of pipelining may grow in future versions of STXXL. We do not show the nonpipelined algorithm for the other inputs since the relative performance compared to pipelined doubling should remain about the same.

A comparison of the new algorithms with previous algorithms is more difficult. The implementation of [Crauser and Ferragina 2002] works only up to 2 GByte of total external memory consumption and would thus have to compete with space efficient internal algorithms on our machine. At least we can compare the I/O volume per byte of input for the measurements in [Crauser and Ferragina 2002]. Their most scalable algorithm for the largest real world input tested (26 MByte of text from the Reuters news agency) is nonpipelined doubling with partial discarding. This algorithm needs an I/O volume of 1303 Bytes per character of input. The DC3-algorithm needs about 5 times less I/Os. Furthermore, it is to be expected that the lead gets bigger for larger inputs. The GBS algorithm [Gonnet et al. 1992] needs 486 bytes of I/O per character for this input in [Crauser and Ferragina 2002], 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 [Crauser and Ferragina 2002] 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.

[8] One might also conclude that a similar increase in internal work could be expected in an implementation of the DC7 algorithm.

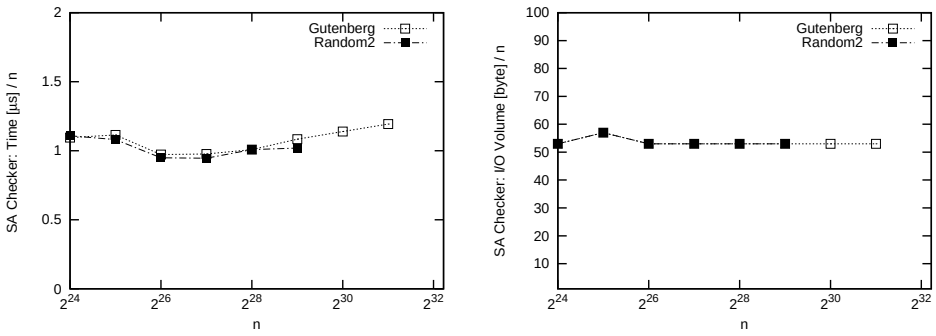



Fig. 12. Execution time (left) and I/O volume (right) for the suffix array checker (Opteron machine).

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

| $D$ | 1 | 2 | 4 | 6 | 8 |
|---|---|---|---|---|---|
| $t[\mu s/\text{byte}]$ | 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 could be used for external suffix array construction.

Figure 11 shows the execution times of the implementations running on the Opteron machine. The implementations need a factor of 1.7–2.4 less time. The largest speedup is observed for the quadrupling with discarding running on the largest source code instance. This might be due to the faster SCSI hard disks with higher bandwidth (70 MB/s versus 45 MB/s) and the shorter seek time (3.6 ms versus 8.8 ms on average), and perhaps a faster 64-bit CPU. However, the relative performance of the algorithms remains the same as in the experiments using the Xeon system.

### The Checker

Figure 12 shows the execution time and the I/O volume of the suffix array checker from Section 8 running on the Opteron system. The horizontal axis denotes the size of the input string $T$. The curves for the other input families are not shown, since the algorithm is not sensitive to the type of input. The implementation onyl needs 1–1.2 $\mu$s per input string character.

## 10. 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 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.

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 overnight on a low cost machine, which is two orders of magnitude more than in [Crauser and Ferragina 2002] in a time faster or comparable to previous internal memory computations [Sadakane and T.Shibuya 2001; Lam et al. 2002] on more expensive machines.

There are also many opportunities to scale to even larger inputs. In Section 7 we have outlined that for small alphabets, the generalized difference cover algorithm DCX, can yield significant further savings in I/O requirements. With respect to internal work, one could exploit that about half of the sorting operations are just permutations. 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 [Kärkkäinen et al. 2006]. An MPI-based [Gropp et al. 1998] distributed memory implementation of DC3 [Kulla and Sanders 2006] scales well up to 128 processors according to the experiments. It looks likely that the algorithm would also scale to thousands of processors. However, the parallel implementation does not use I/O-efficient processing, therefore this leaves room for further improvements which will enable a fast construction of even larger suffix arrays.

## ACKNOWLEDGMENTS

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. The most of the work was done when the authors were at MPII.

## REFERENCES

ABOUELHODA, M. I., KURTZ, S., AND OHLEBUSCH, E. 2002. The enhanced suffix array and its applications to genome analysis. In *Proc. 2nd Workshop on Algorithms in Bioinformatics*. LNCS, vol. 2452. Springer, 449–463.

AGGARWAL, A. AND VITTER, J. S. 1988. The input/output complexity of sorting and related problems. *Commun. ACM 31,* 9, 1116–1127.

ARGE, L., FERRAGINA, P., GROSSI, R., AND VITTER, J. S. 1997. On sorting strings in external memory. In *29th ACM Symposium on Theory of Computing*. ACM Press, El Paso, 540–548.

ARGE, L., PROCOPIUC, O., AND VITTER, J. S. 2002. Implementing I/O-efficient data structures using TPIE. In *Proc. 10th Annual European Symposium on Algorithms*. LNCS, vol. 2461. Springer, 88–100.

BURKHARDT, S. AND KÄRKKÄINEN, J. 2003. Fast lightweight suffix array construction and checking. In *Proc. 14th Annual Symposium on Combinatorial Pattern Matching*. LNCS, vol. 2676. Springer, 55–69.

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

Crauser, A. and Ferragina, P. 1999. On constructing suffix arrays in external memory. In *Proc. 7th Annual European Symposium on Algorithms*. LNCS, vol. 1643. 224–235.

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

Crauser, A. and Mehlhorn, K. 1998. LEDA-SM a platform for secondary memory computations. Tech. rep., MPII. draft.

Dementiev, R. The Stxxl library. Documentation and download at `http://stxxl.sourceforge.net`.

Dementiev, R., Kettner, L., and Sanders, P. 2005. Stxxl: Standard Template Library for XXL Data Sets. In *Proc. 13th Annual European Symposium on Algorithms*. LNCS, vol. 3669. 640–651.

Dementiev, R. and Sanders, P. 2003. Asynchronous parallel disk sorting. In *Proc. 15th Annual Symposium on Parallelism in Algorithms and Architectures*. San Diego, 138–148.

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

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

Gropp, W., Lusk, R., and Thakur, R. 1998. Latest Advances in MPI-2. Tutorial on EuroPVM/MPI'98.

Haanpää, H. 2004. Minimum sum and difference covers of Abelian groups. *Journal of Integer Sequences 7,* 2, article 04.1.8.

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

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

Kärkkäinen, J. 2003. *Algorithms for Memory Hierarchies*. LNCS, vol. 2625. Springer, Chapter Full-Text Indexes in External Memory, 171–192.

Kärkkäinen, J., Sanders, P., and Burkhardt, S. 2006. Linear work suffix array construction. *Journal of the ACM*. to appear.

Kim, D. K., Sim, J. S., Park, H., and Park, K. 2003. Linear-time construction of suffix arrays. In *Proc. 14th Annual Symposium on Combinatorial Pattern Matching*. LNCS, vol. 2676. Springer, 186–199.

Ko, P. and Aluru, S. 2003. Space efficient linear time construction of suffix arrays. In *Proc. 14th Annual Symposium on Combinatorial Pattern Matching*. LNCS, vol. 2089. Springer, 200–210.

Kulla, F. and Sanders, P. 2006. Scalable parallel suffix array construction. In *EuroPVM/MPI*. to appear.

Lam, T.-W., Sadakane, K., Sung, W.-K., and Yiu, S.-M. 2002. A space and time efficient algorithm for constructing compressed suffix arrays. In *Proc. 8th Annual International Conference on Computing and Combinatorics*. LNCS, vol. 2387. Springer, 401–410.

Manber, U. and Myers, G. 1993. Suffix arrays: A new method for on-line string searches. *SIAM J. Comput. 22,* 5 (Oct.), 935–948.

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

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

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

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

Weese, D. 2006. Entwurf und Implementierung eines generischen Substring-Index. M.S. thesis, Freie Universität Berlin.

Received Month Year; revised Month Year; accepted Month Year