Better External Memory Suffix Array Construction

Roman Dementiev*; Juha Kärkkäinen†; Jens Mehnert‡; Peter Sanders*

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 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.

1 Introduction
The suffix array [21, 13], a lexicographically sorted array of the suffixes of a string, has numerous applications, e.g., in string matching [21, 13], 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 [17, 18, 19]. One of the linear time algorithms [17] is very simple and can also be adapted to obtain an optimal algorithm for external memory: The DC3-algorithm [17] constructs a suffix array of a text \( T \) of length \( n \) using \( O(\text{sort}(n)) \) 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 [8] 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 characters.

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 \( \text{SA} \) of \( T \) is a permutation of \([0, n]\) such that \( \text{SA}_i < \text{SA}_j \) 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.

\begin{align}
\maxlcp &:= \max_{0 \leq i < n} \text{lcp}(i, i + 1) \\
\text{lcp} &:= \frac{1}{n} \sum_{0 \leq i < n} \text{lcp}(i, i + 1) \\
\log\text{dps} &:= \frac{1}{n} \sum_{0 \leq i < n} \log(\text{dps}(i))
\end{align}

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 \([\log n]\) 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 \(\text{sort}(x) \approx \frac{x}{DB} \left[ \log_M B \frac{x}{MB} \right]\) 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 using \(\mathcal{S}\) for sequentially read-only \(x\) and \(\mathcal{W}\) for sequentially write-only \(x\). We express all our I/O complexities in terms of \(\text{I/Os for reading the input and writing the output}\).

1.2 Overview. In Section 2 we present the \textit{doubling algorithm} [3, 8] for suffix array construction that has \(\text{I/O complexity } \mathcal{O}(\text{sort}(n) \maxlcp)\). 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 \textit{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. 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 \textit{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 \(\text{I/O complexity } \mathcal{O}(\text{sort}(n \log\text{dps}))\) improving on a previous discarding algorithm with \(\text{I/O complexity } \mathcal{O}(\text{sort}(n \log\text{dps}) + \text{scan}(n \log\maxlcp))\) [8]. 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 (\textit{quadrupling}) or five. A pipelined optimal algorithm with \(\text{I/O complexity } \mathcal{O}(\text{sort}(n))\) in Section 6 concludes our sequence of suffix array construction algorithms.

In Section 7 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 built. The appendices contains further details including A useful tool for testing our implementations was a fast and simple external memory checker for suffix arrays.

1.3 More Related Work. The first I/O optimal algorithm for suffix array construction [12] is based on suffix tree construction and introduced the basic divide-and-conquer approach that is also used by DC3. However, the algorithm from [12] is so complicated that an implementation looks not promising.

There is a very recent study of external suffix tree construction by Crauser and Ferragina [8]. 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. 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 [8] is the GBS-algorithm [13] that takes \(\mathcal{O}(\frac{n}{\text{dps}}\text{scan}(n))\) 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 is a very recent study of external suffix \textit{tree} construction for the case that the text itself fits in internal memory [7]. The proposed algorithm has quadratic worst case complexity. Experiments are reported for up to \(224 \cdot 10^6\) characters (human chromosome 1) and only optimistic estimates of I/O performance rather than actual execution times are given.

There has been considerable interest in space efficient internal memory algorithms for constructing suffix arrays [22, 5] and even more compact full-text in-
consumption is less of an issue because disk space is two
orders of magnitude cheaper than RAM.

The biggest sux array computations we are aware of are
for the human genome [23, 20]. One [20] computes the
compressed sux array on a PC with 3 GBytes of
memory in 21 hours. Compressed sux 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 implementa-
tions [23] uses a supercomputer with 64 GBytes of mem-
ory 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, 9] do not
support it. We decided quite early in the design of
our library STXXL [10] that we wanted to remove this
deficit. Since sux array construction can profit immen-
sely from pipelining and since the different algo-
rithms give a rich set of examples, we decided to use
this application as a test bed for a prototype implement-
tion of pipelining.

2 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 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 $(i \mod 2^k, i \div 2^k)$. 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.

The algorithm performs a constant number of sort-
ing 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.

Function doubling($T$)
$S:= \{(T[i], T[i+1]) : i \in [0,n]\}$ (0)
for $k := 1$ to $\lfloor \log n \rfloor$ do
    sort $S$
    $P:= name(S)$
    invariant $\forall (c,i) \in P$:
        $c$ is a lexicographic name for $T[i,i+2^k]$  
    if the names in $P$ are unique then
        return $[i : (c,i) \in P]$  
    sort $P$ by $(i \mod 2^k, i \div 2^k)$  
    $S:= (((c,c'),i) : j \in [0,n])$
    $(c,i) = P[j], (c',i+2^k) = P[j+1]$ (1)  

Function name($S$ : Sequence of Pair)
$q:= r:= 0$  
foreach $((c,c'),i) \in S$ do
    $q++$
    if $(c,c') \neq (t, t')$ then $r:= q$  
    $(t, t'):= (c,c')$
append $(r,i)$ to result
return result

Figure 1: The doubling algorithm.

Theorem 2.1. The doubling algorithm computes a suf-
fix array using $O(sort(n) \cdot \lfloor \log \max lcp \rfloor)$ 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
digraph $G = (V = F \cup S \cup R, E)$. 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 $b(s)$.

Streaming nodes may cause additional I/Os for internal
processing, e.g., for large FIFO queues or priority queues.
and output it in sorted order. Sorting nodes have a buffer requirement of $b(r) = \Theta(M)$ and outdegree one $^2$. Edges are labeled with the number of machine words $w(e)$ flowing between two nodes. In the proof of Theorem 3.2 you find the flow graph for the pipelined doubling algorithm. We will see a somewhat more complicated graph 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 $w : E \rightarrow \mathbb{R}_+$ and buffer requirements $b : V \rightarrow \mathbb{R}_+$ can be executed using

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

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 $M$.

**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 $G$. $^3$

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 $O(M^2/B)$, 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.

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 $a \cdot \text{scan}(x) = \text{scan}(a \cdot x) + O(1)$ and $a \cdot \text{sort}(x) \leq \text{sort}(a \cdot x) + O(1)$, we can represent the result in the form $\text{scan}(x) + \text{sort}(y) + O(1)$, 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 [8] (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 3.1 to the doubling algorithm:

**Theorem 3.2.** The doubling algorithm from Figure 1 can be implemented to run using $\text{sort}(5n) \cdot \log(1 + \max lcp) + O(\text{scan}(n))$ I/Os.

---

1 I/Os are not counted in our analysis.
2 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.
3 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.
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.

![Flow Graph](image)

After \( [\log (1 + \max lcp)] \) 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 [10]. 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 \texttt{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 creating all node objects, the computation starts in a nodes element by element. Those nodes continue in the function which signals the end of the sequence. After 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 \texttt{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

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 \( c_j^{k+2} < c_i^{k+2} \). 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., \( c_i^j = c_i^k \) 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.

Function \texttt{name2}()

```python
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
```

Figure 3: The alternative naming procedure.

Function \texttt{doubling + discarding}(\( T \))

```python
Function doubling + discarding(T)
S := (((T[i], T[i + 1]), i) : i ∈ [0, n])

sort S
U := name(S) // undiscarded
P := () // partially discarded
F := () // fully discarded

for k := 1 to \lfloor \log n \rfloor do
    mark unique names in U
    sort U by (i \mod 2^k, i \div 2^k)
    merge P into U; P := ()
    S := (); count := 0
    foreach (c, i) ∈ U do
        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', i'), i) to S
                count++
        if S = ∅ then
            sort F by first component
            return [i : (c, i) ∈ F]
    sort S
    U := name2(S)
```

Figure 4: The doubling with discarding algorithm.
c is unique. We will fully discard \((c, i) = (c^k_i, i)\) when also either \(c^k_{i-2} \) or \(c^k_{i-2+i} \) is unique, because then in any iteration \(h > k\), the first component of the tuple \(((c^h_{i-2}, c^h_i), i-2^h)\) must be unique. The final algorithm is given in Figure 4.

**Theorem 4.1.** Doubling with discarding can be implemented to run using \(O(5n \log \text{dps}) + O(\text{sort}(n))\) 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 at most \(N = n \log \text{dps}\) tuples are processed in each sorting step over the whole execution and that at most \(c\) 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 \([\log(1 + \text{dps}(i))])\) 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 [16]. The algorithm in [8] does partial but not full discarding, adding the term \(O(\text{scan}(n \log \text{maxlcp}))\) to its complexity.

### 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.

**Theorem 5.1.** The \(a\)-tupling algorithm can be implemented to run using

\[
\text{sort}(\frac{a+3}{\log a} n \log \text{maxlcp}) + O(\text{sort}(n)) \quad \text{or} \quad \text{sort}(\frac{a+3}{\log a} n \log \text{dps}) + O(\text{sort}(n))
\]

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

The following three step algorithm outlines a linear time algorithm for suffix array construction [17]:

1. Construct the suffix array of the suffixes starting at positions \(i \mod 3 \neq 0\). 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 5 gives pseudocode for an external implementation of this algorithm and Figure 6 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 \(i \mod 3 = 0\) 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 com-
This problem can be solved by writing a temporary copy of the input stream. Note that this is still cheaper than the external DC3 algorithm. Using Theorem 3.1 and the doubling algorithm from Figure 5 we have to read the input once but we save writing it to the local file node \( T \).

We have implemented the algorithms in C++ using the g++ 3.2.3 compiler (optimization level -02 --omit-framepointer) and the external memory library STXXL Version 0.52 [10]. 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 [11] for a performance evaluation of this machine whose cost was 2500 Euro in July 2002. The following instances have been considered:

**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

\[
\begin{align*}
T_{\text{mod-0 sux}} &= T_{\text{mod-2 sux}} + 2.0 \text{ GHz Intel Xeon processors, one GByte of RAM, and we use four 80 GByte IBM 120GXP disks.}
\end{align*}
\]

**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.

---

**Theorem 6.1**: The doubling algorithm from Figure 5 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 3.1 and the data flow diagram from Figure 6 we can conclude that

\[
\begin{align*}
V(n) &\leq \text{sort}((n/3 + n/3 + n/3 + n/3)) + \text{sort}(2n) + \text{scan}(2n) + V(\frac{2}{3}n) \\
&= \text{sort}(10n) + \text{scan}(2n) + V(\frac{2}{3}n)
\end{align*}
\]

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 \( T \).

**Figure 5**: The DC3-algorithm.

---

**Function DC3(T)**

\[
\begin{align*}
S &:= \{[(T[i, i + 2]), i) : i \in [0, n), i \mod 3 \neq 0 \} \\
\text{sort } S &\text{ by the first component} \\
P &:= \text{name}(S)
\end{align*}
\]

if the names in \( P \) are not unique then

\[
\begin{align*}
sort &\text{ the }(i, r) \in P \text{ by } (i \mod 3, i \div 3) \\
\text{SA}^{12} &:= \text{DC3}([c : (c, i) \in P]) \\
P &:= \{[j + 1, \text{SA}^{12}[j]) : j \in [0, 2n/3] \}
\end{align*}
\]

sort \( P \) by the second component

\[
\begin{align*}
S_0 &:= \langle[T[i], T[i + 1], c', c''], i \rangle : \\
i \mod 3 &= 0, (c', i + 1), (c'', i + 2) \in P
\end{align*}
\]

\[
\begin{align*}
S_1 &:= \langle(c, T[i]), c', i) : \\
i \mod 3 &= 1, (c, i), (c', i + 1) \in P
\end{align*}
\]

\[
\begin{align*}
S_2 &:= \langle(c, T[i], T[i + 1], c'', i) : \\
i \mod 3 &= 2, (c, i), (c'', i + 2) \in P
\end{align*}
\]

sort \( S_0 \) by components 1,3

sort \( S_1 \) and \( S_2 \) by component 1

\[
\begin{align*}
S &:= \text{merge}(S_0, S_1, S_2) \text{ comparison function:} \\
(t, t', c', c'', i) &\in S_0 \Rightarrow (t', c') \leq (u, d') \\
(t, t', c', c'', i) &\in S_0 \Rightarrow (d, u, u', d'', j) \in S_2 \\
\Rightarrow (t, t', c', c'') &\leq (u, u', d''), \\
(c, t, c', i) &\in S_1 \Rightarrow (d, u, u', d'', j) \in S_2 \\
\Rightarrow c &\leq d
\end{align*}
\]

return [last component of \( s : s \in S \)]
Figure 6: Data flow graphs for the DC3-algorithm. The numbers refer to line numbers in Figure 5.

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

|        | $n = |T|$ | $|T| \Sigma$ | maxlcp | $\overline{lcp}$ | log dps |
|--------|--------|-------------|--------|------------------|--------|
| Random2| $2^{42}$ | 128         | $2^{41}$ | $\approx 2^{39}$ | $\approx 29.56$ |
| 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 |

**Source:** Source code (mostly C++) containing core-utils, 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 $T[0, 2^n]$. 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, 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 dps.

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 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 our implementation. 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 [8] 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 [8]. 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 [13] needs 486 bytes of I/O per character for this input in [8], 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 [8] 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.

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

<table>
<thead>
<tr>
<th>D</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>6</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>t[µs/byte]</td>
<td>13.96</td>
<td>9.88</td>
<td>8.81</td>
<td>8.65</td>
<td>8.52</td>
</tr>
</tbody>
</table>

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.

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 7 and is based on the following result.

**Lemma 8.1.** ([5]) An array $SA[0, n)$ is the suffix array of a text $T$ if the following conditions are satisfied:

1. $SA$ contains a permutation of $[0, n)$.
2. $\forall i, j : r_i \leq r_j \iff (T[i], r_{i+1}) \leq (T[j], r_{j+1})$ 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. \hfill \Box

**Theorem 8.1.** The suffix array checker from Figure 7 can be implemented to run using $\text{sort}(5n) + \text{scan}(2n)$ I/Os.

**Function Checker(SA, T)**

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

Figure 7: The suffix array checker.

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 overnight on a low cost machine. Two orders of magnitude more than in [8] 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 [17]. 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 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.

References


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.