Scalable Construction of Text Indexes with Thrill

Timo Bingmann, Simon Gog
Institute of Theoretical Informatics,
Karlsruhe Institute of Technology, Germany
Email: firstname.lastname@kit.edu

Florian Kurpicz
Department of Computer Science,
Technische Universität Dortmund, Germany
Email: florian.kurpicz@tu-dortmund.de

***Abstract*—The suffix array is the key to efficient solutions for myriads of string processing problems in different application domains, like data compression, data mining, or bioinformatics. With the rapid growth of available data, suffix array construction algorithms have to be adapted to advanced computational models such as external memory and distributed computing. In this article, we present five suffix array construction algorithms utilizing the new algorithmic big data batch processing framework Thrill, which allows scalable processing of input sizes on distributed systems in orders of magnitude that have not been considered before.**

***Keywords*-suffix array; C++; big data tool; distributed data processing;**

Suffix arrays [1], [2] are the basis for many text indices and string algorithms. Suffix array construction is theoretically linear work, but practical suffix sorting is computationally intensive and often limits the applicability of advanced text data structures on large datasets. While fast sequential algorithms exist in the RAM model [3], [4], these are limited by the CPU power and RAM size of a single machine. External memory algorithms on a single machine are limited by disk space [5]–[8], and often have long running times due to mostly sequential computation or limited I/O bandwidth.

Most suffix array construction algorithms (SACAs) focus only on sequential computation models. However, while the volume of data is increasing, the speed of individual CPU cores is not. This leaves us no choice but to consider shared-memory parallelism and distributed cluster computation to gain considerable speedups in the future. For this reason, we propose to use the new big data processing framework *Thrill* [9] to implement *distributed external memory* suffix sorting algorithms for big data inputs.

Most SACAs employ a subset of three basic suffix sorting principles: *prefix doubling*, *recursion* and *inducing* [10]. The last type, *inducing*, is the basis for the fastest *sequential* suffix array construction algorithms [3], [4]. It yields however only well to parallelization for small alphabets [11] and it is unclear how to map induced sorting efficiently to distributed environments. Recently, a fast distributed prefix doubling implementation using MPI has been presented [12]. While they report high speeds for small inputs, we could not successfully run their implementation on large inputs.

Our first three algorithms in Thrill are based on prefix doubling. In Section II we first review the main idea behind this technique, and then show how to implement it using only the scalable primitives provided by Thrill. The result are prefix doubling *using the inverse suffix array* (Section II-A), prefix doubling *using sorting* (Section II-B), and prefix doubling *with discarding* (Section II-C).

Our last two algorithms are distributed formulations of the linear-time difference cover algorithm *DC* [13]. These employ sorting, recursion, prefix sums, and merging of arrays, which are all scalable primitives in Thrill. In Section III we recall the DC3 algorithm in abstract form, and then present DC3 and DC7 in Section III-A using concrete Thrill pseudocode.

In Section IV we run the five Thrill implementations on up to 32 hosts with fast NVMe SSDs and limited RAM in the AWS Elastic Compute Cloud (EC2). We compare them against two independent MPI implementations and the two fastest non-distributed sequential suffix sorters as a baseline. Our Thrill implementations scale higher than the MPI implementations, which are constrained by RAM and fail after 2 GiB. Comparing to the fastest sequential suffix sorters, our best Thrill implementations outperform on digits of π when run with 2 hosts (32 cores), and on Wikipedia when run with 4 hosts (64 cores).

I. Related Work

While there exist numerous works on sequential suffix array construction, there is much fewer works on distributed suffix sorting and no publications using distributed external memory. In this paper we only review the publications most relevant for our implementations in Thrill. Please refer to our longer version [14, ch. 5] for a discussion of previous external and distributed suffix sorting algorithms.

In 2003, Kärkkäinen and Sanders [15] presented a linear-time SACA, the so called DC3 algorithm, that works well in multiple advanced models of computation such as external memory [5]–[8] and also parallel and distributed environments. Kulla and Sanders later demonstrated the scalability of the DC3 algorithm [16] by presenting a MPI implementation. More recently, Flick and Aluru presented an implementation of a prefix doubling algorithm in MPI that can also compute the longest common prefix array [12]. Suffix array construction has also been considered in external memory, where in theory the DC3 algorithm is optimal. Dementiev, Kärkkäinen, Mehnert, and Sanders [6] compared

multiple variants of prefix doubling and DC3 for external memory in practice. Our prefix doubling and difference cover algorithm implementations in Thrill are based on these preceding publications.

As induced sorting has been successfully redesigned for external memory [17], [18], the question naturally arises why we did not consider it in a distributed environment. The problem is that induced sorting appears very difficult to parallelize. Up to now, it yields only well to parallelization for specific inputs on small alphabets [11], and hence does not appear to be a promising approach for general inputs in a highly parallel and distributed setting. However, more future work in this direction is needed, as important applications such as bioinformatics have small alphabets.

A. Preliminaries and a Short Introduction into Thrill

Let $[0,n] := \{0,\ldots,n\}$ and $[0,n) := \{0,\ldots,n-1\}$ be ranges of integers. For any array A, we write $A[\ell,r]$ (or $A[\ell,r)$) to denote the sub-array of A ranging from ℓ to r (or $r-1$). A string $T = [t_0,\ldots,t_{|T|-1}]$ consists of $|T|$ characters from a totally ordered alphabet Σ. For simplicity, we assume that $t_{|T|-1}$ is a unique character '$' that is also the lexicographically smallest. We call the substring $T[i..|T|)$ the i-th *suffix* of T. The *suffix array* (SA) of T is a permutation of $[0,|T|)$ such that $T[\mathsf{SA}[i]..|T|) < T[\mathsf{SA}[j]..|T|)$ for all $0 \le i < j < |T|$. We call the inverse permutation of SA the *inverse suffix array* (ISA). The *longest common prefix* of two suffixes is $\text{LCP}(i,j) := \max\{s\colon T[i..i+s) = T[j..j+s)\}$ and the maximum length of any common prefix is denoted by $\text{maxlcp} := \max\{\text{LCP}(i,j)\colon 0 \le i < j < |T|\}$.

We implemented five SACAs using the distributed big data batch computation framework *Thrill* [9], which works with *distributed immutable arrays* (DIAs) storing tuples. Items in DIAs cannot be accessed directly, instead there is a rich set of DIA operations which can be used to transform DIAs (we use and describe only a subset of these operations).

$\mathbf{Filter}(\boldsymbol{f})$ takes a DIA$\langle A\rangle$ X and a function $f\colon A \to \texttt{bool}$, and returns the DIA$\langle A\rangle$ containing $[\,x \in X \mid f(x)\,]$ within which the order of items is maintained.

$\mathbf{Map}(\boldsymbol{f})$ applies the function $f\colon A \to B$ to each item in the input DIA$\langle A\rangle$ X, and returns a DIA$\langle B\rangle$ Y with $Y[i] = f(X[i])$ for all $i = 0,\ldots,|X|-1$.

$\mathbf{Window}_{\boldsymbol{k}}(\boldsymbol{w})$ and $\mathbf{FlatWindow}_{\boldsymbol{k}}(\boldsymbol{w'})$ take an input DIA$\langle A\rangle$ X and a window function $w\colon \mathbb{N}_0 \times A^k \to B$. The operation scans over X with a window of size k and applies w once to each set of k consecutive items from X and their index in X. The final $k-1$ indexes with less than k consecutive items are delivered to w as partial windows padded with sentinel values. The result of all invocations of w is returned as a DIA$\langle B\rangle$ containing $|X|$ items in the order. FlatWindow is a variant of Window which takes a input DIA$\langle A\rangle$ X and a window function $w'\colon \mathbb{N}_0 \times A^k \to \text{list}(B)$, who can *emit* zero or more items that are concatenated in the resulting DIA$\langle B\rangle$.

$\mathbf{PrefixSum}(\boldsymbol{s})$: Given an input DIA$\langle A\rangle$ X and an associative operation $s\colon A \times A \to A$ (by default $s = +$), PrefixSum returns a DIA$\langle A\rangle$ Y such that $Y[0] = X[0]$ and $Y[i] = s(Y[i-1], X[i])$ for all $i = 1,\ldots,|X|-1$.

$\mathbf{Sort}(\boldsymbol{c})$ sorts an input DIA$\langle A\rangle$ X with respect to a less-comparison function $c\colon A \times A \to \texttt{bool}$.

$\mathbf{Merge}(\boldsymbol{X_1,\ldots,X_n,c})$: Given a set of sorted DIA$\langle A\rangle$s $X_1,\ldots,X_n$ and a less-comparison function $c\colon A \times A \to \texttt{bool}$, Merge returns DIA$\langle A\rangle$ Y that contains all tuples of $X_1,\ldots,X_n$ and is sorted with respect to c.

$\mathbf{Zip}(\boldsymbol{X_1,\ldots,X_n,f})$: Given a set of DIAs $X_1,\ldots,X_n$ of type $A_1,\ldots,A_n$ of equal size ($|X_1| = \cdots = |X_n|$) and a function $f\colon A_1 \times \cdots \times A_n \to B$, Zip returns DIA$\langle B\rangle$ Y with $Y[i] = f(X_1[i],\ldots,X_n[i])$ for all $i = 0,\ldots,|X_1|-1$.

$\mathbf{ZipWithIndex}(\boldsymbol{f})$: Given an input DIA$\langle A\rangle$ X and a function $f : A \times \mathbb{N}_0 \to B$, ZipWithIndex returns DIA$\langle B\rangle$ Y with $Y[i] = f(X[i], i)$ for all $i = 0,\ldots,|X|-1$

$\mathbf{ZipWindow}_{[\boldsymbol{k_1,\ldots,k_n}]}(\boldsymbol{[X_1,\ldots,X_n],z})$: Combines a Zip operation and a Window operation. Given a list DIA$\langle A_1\rangle X_1$, ..., DIA$\langle A_n\rangle X_n$, and a function $z : \mathbb{N}_0 \times A_1^{k_1} \times \ldots \times A_n^{k_n} \to B$, ZipWithIndex returns DIA$\langle B\rangle$ Y with $Y[i] = z(i, X_1[i,\ldots,i+k_1],\ldots,X_n[i,\ldots,i+k_n])$ for all $i = 0,\ldots,|X|-1$

$\mathbf{Max}(\boldsymbol{c})$: Given an input DIA$\langle A\rangle$ X, Max returns the maximum item $m = \max_c X$ with respect to a less-comparison function $c\colon A \times A \to \texttt{bool}$.

$\mathbf{Size}()$: Given an input DIA$\langle A\rangle$ X, Size returns the number of items in X, i.e., $|X|$.

Thrill applies chains of functions (*method chaining*) to a DIA, e.g., if we have a DIA$\langle \mathbb{N}_0\rangle$ $N = \{0,1,2,\ldots,9\}$ and want to compute the prefix sum of all odd elements, then we write N.Filter($a \mapsto (a \mod 2) = 1$).PrefixSum().

II. Prefix Doubling Algorithms

In this section, we start by reviewing *prefix doubling* for suffix sorting. For better exposition of these ideas, we define the *h-order* $\le_h$ on strings as their lexicographic order limited to depth h: $a|_h$ and $b|_h$ are the string a or b truncated to h characters. Then $a \le_h b$ if and only if $a|_h \le b|_h$. Other comparison operators like $a =_h b$ and $a <_h b$ are defined accordingly. For $h < |T|$ the h-order of suffixes of a string T may not be unique, e.g. with respect to $\le_2$ two suffixes starting with the same 2 characters are considered equal, their order is not fixed. A set of suffixes equal under $=_h$ is called an *h-group* and they all start with the same h characters. A *rank* with respect to $\le_h$ of the suffixes in an h-group is any number greater than the total size of all h-groups containing lexicographically smaller suffixes and smaller than any rank of an h-groups containing lexicographically larger suffixes. A rank with respect to $\le_h$ is also called a *(lexicographic) h-name* or *h-rank*.

Since h doubles in each round, $h \ge |T|$ after $\lceil \log_2 |T| \rceil$ rounds and thus prefix doubling algorithms have worst case running time $\mathcal{O}(|T| \log |T|)$. To be more precise,

Algorithm 1: Prefix Doubling Algorithm in Thrill.

```
1  Function PrefixDoubling(T ∈ DIA⟨Σ⟩)
2      S := T.Window_2((i, [t_0, t_1]) ↦ (i, t_0, t_1))
3      for k := 1 to ⌈log_2 |T|⌉ − 1 do
4          S := S.Sort((i, n_0, n_1) by (n_0, n_1))
5          N := S.FlatWindow_2((j, [a, b]) ↦ CName(j, a, b))
6          if N.Filter((i, n) ↦ (n = 0)).Size() = 1 then
7              return N.Map((i, n) ↦ i)
8          N := N.PrefixSum((i, n), (i′, n′) ↦ (i′, max(n, n′))
9          S := Generate new name pairs using N
10 Function CName(j ∈ ℕ_0, (i, n_0, n_1), (i′, n′_0, n′_1) ∈ N)
11     if j = 0 then emit (i, 0)
12     if (n_0, n_1) ≠ (n′_0, n′_1) then emit (i′, j)
13     else emit (i′, 0)
```

the algorithm terminates when SA^h has no more unsorted h-groups and becomes the suffix array. This already happens after $\lceil \log_2(\text{maxlcp}(T)) \rceil$ iterations, yielding $\mathcal{O}(n \log(\text{maxlcp}(T)))$ running time.

The essential goal of a prefix doubling algorithm is to give each suffix a lexicographic 2^k-name in iteration k using information from iteration $k-1$. Manber and Myers [1] observed that one can compute a 2^k-name for the prefix $T[i..i+2^k)$ of suffix $T[i..|T|)$ using already computed 2^{k-1}-names of the prefixes $T[i..i+2^{k-1})$ and $T[i+2^{k-1}..i+2^k)$. The main idea to bringing this to external and distributed memory is to store and sort tuples (i, n_i) containing names n_i in such a way that we gain triples $(i, n_i, n_{i+2^{k-1}})$ in iteration k. Using two such triples one can take the step from 2^{k-1}-names to 2^k-names: consider $(i, n_i, n_{i+2^{k-1}})$ and $(j, n_j, n_{j+2^{k-1}})$ with $n_i = n_j$. This means that suffixes i and j start with the same 2^{k-1} characters, $T[i..|T|) =_{(2^{k-1})} T[j..|T|)$. By comparing $n_{i+2^{k-1}}$ and $n_{j+2^{k-1}}$, we can determine the lexicographic order of the next 2^{k-1} characters, and hence compute new names.

Algorithm 1 describes the basic structure of the prefix doubling algorithms in Thrill presented in this section. The whole algorithm requires one DIA N storing pairs and one DIA S storing triples. For the first iteration, S contains the triples $(i, T[i], T[i+1])$ for all $i = 0, \ldots, |T|-1$ (line 2). These triples contain a text position and the *name pair* for that position, i.e, the two names that are required to compute the new name for the suffix starting at the text position. For bootstrapping the first iteration $k=1$, we can simply use the characters as 1-names. In our actual implementation, we accelerated the first iteration with *alphabet compression*. Quite often the input string does not use the whole alphabet range. Hence, one can reduce the alphabet size by first counting how many distinct characters occur in the input, and then monotonously mapping them to a compressed range $[0 \mathinner{..} |\Sigma|)$. The input string and the mapped string have the same suffix array, as the lexicographic order of suffixes does not change. In our implementation, we then *pack* as many mapped characters as possible into an integer index.

For subsequent iterations, we continue on line 4 and sort S with respect to the name pair, which brings *equal* 2^{k-1}-names together. These entries with equal 2^{k-1}-name need to be extended to prefix depth 2^k. The new 2^k-names are calculated using a FlatWindow_2 on S (line 5) and the function CName(), which takes the current position i in S and the items $S[i]$ and $S[i+1]$ as input and emits a tuple consisting of a text position and a new name (line 10). We know that the suffixes are sorted with respect to their name pairs. Therefore, we can scan S and mark every position where the name pair differs from its predecessor. CName() marks these non-unique name pairs by giving them the name 0. All unique name pairs get a name equal to their current position in S. If there is only one suffix with name 0, then we know that all names differ and that we have finished the computation, see line 6. Otherwise, we can use the DIA operation PrefixSum() with a *max* operator to set the name of each tuple to the largest preceding name (line 8). The sequence of names is initialized by emitting an arbitrary first name (zero in line 11 of line 10) as first item in the DIA. With this extra item, the name array N always contains $|T|$ items. Now each suffix has a new, more refined name.

The next step (line 9) is to identify the ranks of the suffixes required for the next doubling step. During the k-th doubling step, we fill S with one triple for each index $i = 0, \ldots, |T|-1$ that contains the current name of the suffix at position i and the current name of the suffix at position $i + 2^{k-1}$. For this step, we propose two different approaches: one using the inverse suffix array and a Window operation (Section II-A), and one using sorting (Section II-B).

A. Generating Names using the Inverse Suffix Array

We can obtain the h-names of the required suffixes using the inverse h-suffix array. This approach is based on the qsufsort algorithm [19] and was pioneered in a distributed setting using MPI by Flick and Aluru [12].

Algorithm 2 shows in line 2 how we can sort pairs in N based on their position in the text, such that we get the inverse 2^k-suffix array ISA^{2^k} in iteration k. This inverse 2^k-suffix array contains the current 2^k-name of each suffix. For each position i, we need the name of the $(i+2^k)$-th suffix. To get this name, we can simply scan over the DIA N with a Window() operation of width 2^k+1, i.e., the same as shifting the inverse 2^k-suffix array by 2^k positions and appending 0s until its length is $|T|$.

While our experiments show that this approach is faster than prefix doubling using sorting (described in the next subsection), it is obvious that it only works as long as the Window() size 2^k+1 fits into the RAM of each worker.

Algorithm 2: Generating Names using the ISA in Thrill.

```
1 Function PrefixDoublingISAWindow(k ∈ ℕ0)
2   N := N.Sort((i, n) by i)
3   S := N.Window_{2^k+1}((j, [(i, n), ..., (i', n')]) ↦
        { (i, n, n')  if j + 2^k < |T| ,
        { (i, n, 0)   otherwise.
```

$$S := N.\text{Window}_{2^k+1}\big((j, [\,(i,n), \ldots, (i',n')\,]) \mapsto \begin{cases}(i,n,n') & \text{if } j+2^k < |T|\,,\\ (i,n,0) & \text{otherwise.}\end{cases}\big)$$

Algorithm 3: Generating Names using Sorting in Thrill.

```
1 Function PrefixDoublingSorting(N, k ∈ ℕ0)
2   N := N.Sort(<^k_op)
3   S := N.Window_2((j, [(i, n0, n1), (i', n'0, n'1)]) ↦
        { (i, n0, n'0)  if i + 2^k = i' ,
        { (i, n0, 0)    otherwise.
```

$$S := N.\text{Window}_{2}\big((j, [\,(i,n_0,n_1), (i',n_0',n_1')\,]) \mapsto \begin{cases}(i,n_0,n_0') & \text{if } i+2^k = i'\,,\\ (i,n_0,0) & \text{otherwise.}\end{cases}\big)$$

The second solution using sorting does not suffer from this limitation and can be used as a fallback method.

B. Generating Names using Sorting

Next, we adapt an external memory prefix doubling algorithm by Crauser and Ferragina [5] to Thrill. The idea is to compute the new name pairs by sorting the old names with respect to the starting position of the suffix, as shown in Algorithm 3. During each iteration we know for each suffix the suffix index whose current name is required to compute the next refined name. Hence, we can sort the tuples containing the starting positions of the suffixes and their current name in such a way that if there is another name required for a name pair, then it is the name of the succeeding tuple (line 2). To do so, we use the following comparison operator: $<^k_{\text{op}}$: $(\mathbb{N}_0, \mathbb{N}_0) \times (\mathbb{N}_0, \mathbb{N}_0) \to$ `bool` in Algorithm 3:

$$(i, n) <^k_{\text{op}} (i', n') = \begin{cases} i \textbf{ div } 2^k < i' \textbf{ div } 2^k & \text{if } i \equiv i' \pmod{2^k}\,,\\ i \textbf{ mod } 2^k < i' \textbf{ mod } 2^k & \text{otherwise.}\end{cases}$$

This relation orders pairs (i, n) first by the k least significant bits and then by the $w - k$ most significant bits of i, where w is the number of bits used to store i. For example $<^2_{\text{op}}$ reorders $[0\,..\,8)$ to $[\,0, 4, 1, 5, 2, 6, 3, 7\,]$.

After sorting using the $<^k_{\text{op}}$-comparator, we need to ensure that two consecutive names are the ones required to compute the new name, since the required name may not exist due to the length of the text. This occurs during the k-th iteration for each suffix beginning at a text position greater than $|T| - 2^k$. In this case we use the sentinel name 0, which compares smaller than any valid name (line 3). We return one triple for each position, consisting of a text position, the current name of the suffix beginning at that position and the name of the suffix 2^k positions to the right (if it exists and 0 otherwise).

C. Distributed External Prefix Doubling with Discarding

Both prefix doubling variants presented in the previous two sections have large I/O costs from repeatedly re-ranking suffixes whose final rank is already known. These are included in each distributed sorting operation and cause needless overhead. Crauser and Ferragina [5], and Dementiev, Kärkkäinen, Mehnert and Sanders [6] presented a method called *discarding* to alleviate this by omitting all suffixes no longer needed from sorting operations. To this end, we classify suffixes into three categories:

1) Suffixes that do not yet have a unique name are called *not unique*, which is also the initial state,
2) suffixes that have a unique name, but are required to compute another name pair for a suffix that does not yet have a unique name, are called *unique*, and finally
3) unique suffixes that are no longer needed for any other can be *discarded*.

Using this classification we can extend the prefix doubling algorithm using sorting for generating new names (Section II-B) to exclude unique and discarded suffixes from expensive sorting operations. To be more precise, we can ignore discarded suffixes during all sorting operations, and unique suffixes are only required during the computation of the new names (Algorithm 3). Here, they are needed as second rank of the triple (n'_0 in line 3). We do not emit a triple for an index corresponding to a unique suffix. Instead, we just store the unique pair. The algorithm terminates when all suffixes are either unique or discarded. To compute the final suffix array, we concatenate the unique and discarded pairs and sort them by their rank.

Since we ignore discarded suffixes during the computation of the new ranks, we must compute the new rank based on the old rank instead of on the position among all other suffixes, as we did before. This can be done using multiple prefix sum operations. In addition, we must keep track of the unique and discarded suffixes, but the total overhead is small compared to the savings during the sorting operations.

Please refer to our longer version [14, ch. 8.2.5] for the pseudocode and full explanation of the discarding algorithm.

III. Difference Cover Algorithms

In 2003, the *skew* aka *DC3* suffix sorting algorithm was proposed by Kärkkäinen and Sanders [15], and later generalized to *DC* by Kärkkäinen, Sanders, and Burkhardt [13]. They employ recursion on a subset of the suffixes to reach linear running time in the sequential RAM model. The algorithms were later implemented for external memory [6], [20], and DC3 for distributed memory using MPI [16].

The key notion of DC is to recursively calculate the ranks of suffixes in only a *difference cover* [21] of the original text. A set $D \subseteq \mathbb{N}_0$ is a difference cover for $v \in \mathbb{N}_0$, if $\{(i - j) \bmod v \mid i, j \in D\} = \{0, \ldots, v - 1\}$. Examples of

difference covers are $D_3 = \{1, 2\}$ for $v = 3$, $D_7 = \{0, 1, 3\}$ for $v = 7$, and $D_{13} = \{0, 1, 3, 9\}$ for $v = 13$. In general, a difference cover of size $\mathcal{O}(\sqrt{v})$ can be calculated for any v in $\mathcal{O}(\sqrt{v})$ time [13]. With respect to suffix sorting, the difference cover has the interesting property that it *samples* suffixes for recursive sorting and given the rank of all samples allows one to order the non-sample suffixes using a *constant-time* comparison operation. The basic steps of the DC3 algorithm are the following:

D1) Calculate ranks for all suffixes starting at positions in the difference cover $D_3 = \{1, 2\}$ modulo 3. This is done by sorting the triples $(T[i], T[i + 1], T[i + 2])$ for $(i \bmod 3) \in D_3$, calculating lexicographic names, sorting the names back to string order, and recursively calling a suffix sorting algorithm on a reduced string T_R of size $\frac{2}{3}|T|$, if necessary. This reduced string represents *two* concatenated copies of the input string using the lexicographic names: the first copy are all names for suffixes with $i = 1 \bmod 3$ followed by a second copy for all suffixes with $i = 2 \bmod 3$. Hence, each character in T_R embodies three characters in T. Step D1) calculates two arrays, R_1 and R_2, containing the ranks of suffixes $i = 1 \bmod 3$ and $i = 2 \bmod 3$, which are computed by inverting the recursively constructed suffix array of T_R.

D2) Scan the text T and rank arrays R_1 and R_2 to generate three arrays: S_0, S_1, and S_2, where array S_j contains one tuple for each suffix i with $i = j \bmod 3$. For each suffix i, the arrays store one tuple containing the two *following* ranks from R_1 and R_2 and all characters from T *up to* the next ranks. This is exactly the information required such that the following merge step is able to deduce the suffix array correctly. Due to the difference cover property the following rank for each suffix i is among the three elements $R_1[i]$, $R_1[i+1]$, and $R_1[i+2]$ for R_1, and analogously for R_2.

D3) Sort S_0, S_1, and S_2 and merge them using a custom comparison function which compares the suffixes represented in the tuples using characters and ranks. Only a constant number of characters and ranks need to be accessed in each comparison. Output the suffix array using the indices stored in tuples.

The first two steps of the DC3 algorithms can be seen as preparation for the final merge in step D3). Step D1) delivers ranks for all suffixes $(i \bmod 3) \in D_3$ in R_1 and R_2. In step D2) tuples are created in S_0, S_1, and S_2 which are constructed from the recursively calculated ranks and characters from the text. The tuples are designed such that the comparison function can fully determine the final suffix array.

The DC3 algorithm generalizes to *DC* using a difference cover D for any ground set size $v \geq 3$. DC constructs a recursive subproblem of size $\lceil(|T|/v)|D|\rceil$, which, considering $|D| = \mathcal{O}(\sqrt{v})$, is of size $\Theta(\frac{1}{\sqrt{v}})$. The algorithm has at most $\log_v |T|$ recursion levels and only one recursion branch. At every level of the recursion, only work with sorting complexity is needed, and a straightforward application of the Master theorem to the recurrence $Z(|T|) = Z(\Theta(\frac{|T|}{\sqrt{v}})) + \mathcal{O}(\text{sort}(|T|))$ shows that the whole algorithm has sorting complexity due to the small recursive subproblem. For our distributed scenario, DC has the same complexity as sorting and merging.

Algorithm 4: DC3 Algorithm in Thrill.

1 **Function** DC3($T \in \text{DIA}\langle\Sigma\rangle$)
2 $T_3 := T$.FlatWindow$_3$($(i, [\,c_0, c_1, c_2\,]) \mapsto$ Triple(i, c_0, c_1, c_2))
3 **with Function** Triple($i \in \mathbb{N}_0,\ c_0, c_1, c_2 \in \Sigma$)
4 **if** $i \neq 0 \bmod 3$ **then** **emit** (i, c_0, c_1, c_2)
5 $S := T_3$.Sort((i, c_0, c_1, c_2) by (c_0, c_1, c_2))
6 $I_S := S$.Map($(i, c_0, c_1, c_2) \mapsto i$)
7 $N' := S$.FlatWindow$_2$($(i, [\,p_0, p_1\,]) \mapsto$ CTriple(i, p_0, p_1))
8 **with Function** CTriple($i \in \mathbb{N}_0,\ (c_0, c_1, c_2),\ (c'_0, c'_1, c'_2)$)
9 **if** $i = 0$ **then** **emit** 0
10 **emit** (**if** $(c_0, c_1, c_2) = (c'_0, c'_1, c'_2)$ **then** 0 **else** 1)
11 $N := N'$.PrefixSum()
12 $n_{\text{sub}} = \lceil 2|T|/3 \rceil$, $n_{\text{mod1}} = \lceil |T|/3 \rceil$
13 **if** N.Max() $+ 1 \neq n_{\text{sub}}$ **then**
14 $T'_R :=$ Zip($[\,I_S, N\,], (i, n) \mapsto (i, n)$) .Sort($(i, n)$ by $(i \bmod 3, i \text{ div } 3)$)
15 $\mathsf{SA}_R :=$ DC3(T'_R.Map($(i, n) \mapsto n$))
16 $I'_R := \mathsf{SA}_R$.ZipWithIndex($(r, i) \mapsto (r, i)$)
17 $I_R := I'_R$.Sort((r, i) by $(r \bmod n_{\text{mod1}}, r)$)
18 **else**
19 $R := I_S$.ZipWithIndex($(r, i) \mapsto (r, i)$)
20 $I_R := R$.Sort((r, i) by $(r \text{ div } 3, r)$)
21 $I_R := I_R$.Map($(r, i) \mapsto (i + 1)$)
22 $Z' :=$ ZipWindow$_{[3,2]}$($[T, I_R]$, $(i, [\,c_0, c_1, c_2\,], [\,r_1, r_2\,]) \mapsto (c_0, c_1, c_2, r_1, r_2)$)
23 $Z := Z'$.Window$_2$($(i, [\,(z_1, z_2)\,]) \mapsto (i, z_1, z_2)$)
24 $S_0 := Z$.Map($(i, (c_0, c_1, c_2, r_1, r_2), (\bar{c}_0, \bar{c}_1, \bar{c}_2, \bar{r}_1, \bar{r}_2)) \mapsto (3i + 0, c_0, c_1, r_1, r_2)$).Sort($(i, c_0, c_1, r_1, r_2)$ by (c_0, r_1))
25 $S_1 := Z$.Map($(i, (c_0, c_1, c_2, r_1, r_2), (\bar{c}_0, \bar{c}_1, \bar{c}_2, \bar{r}_1, \bar{r}_2)) \mapsto (3i + 1, c_1, r_1, r_2)$).Sort($(i, c_1, r_1, r_2)$ by (r_1))
26 $S_2 := Z$.Map($(i, (c_0, c_1, c_2, r_1, r_2), (\bar{c}_0, \bar{c}_1, \bar{c}_2, \bar{r}_1, \bar{r}_2)) \mapsto (3i + 2, c_2, r_2, \bar{c}_0, \bar{r}_1)$).Sort($(i, c_2, r_2, \bar{c}_0, \bar{r}_1)$ by (r_2))
27 **return** Merge($[\,S_0, S_1, S_2\,]$, CmpDC3).Map($(i, \ldots) \mapsto i$)
with Function CmpDC3(z_1, z_2)
28 $(c_0, r_1) < (c'_1, r'_2)$ if $z_1 = (i, c_0, c_1, r_1, r_2) \in S_0$, $z_2 = (i', c'_1, r'_1, r'_2) \in S_1$,
29 $(c_0, c_1, r_2) < (c'_2, \bar{c}'_0, \bar{r}'_1)$ if $z_1 = (i, c_0, c_1, r_1, r_2) \in S_0$, $z_2 = (i', c'_2, r'_2, \bar{c}'_0, \bar{r}'_1) \in S_2$,
30 $(r_1) < (r'_2)$ if $z_1 = (i, c_1, r_1, r_2) \in S_1$, $z_2 = (i', c'_2, r'_2, \bar{c}'_0, \bar{r}'_1) \in S_2$,
31 and symmetrically if $z_1 \in S_i, z_2 \in S_j$ with $i > j$.

A. Distributed Difference Cover Algorithms with Thrill

The complete DC3 implementation in Thrill algorithm code is shown as Algorithm 4. In the algorithm pseudocode we omitted some details on padding and sentinels for inputs that are not a multiple of the difference cover size, but our actual implementation in Thrill covers all these edge cases.

Goal of lines 2–20 is to calculate R_1 and R_2 as an interleaved array I_R. This is done by performing the following steps:

1) Scan the text T using a FlatWindow operation and create triples (i, c_0, c_1, c_2) for all indices $(i \bmod 3) \in D_3 = \{1, 2\}$ (lines 2–4).
2) Sort the triples as S, scan S and use a prefix sum to calculate lexicographic names N (lines 5–11). The lexicographic names are constructed in the prefix sum from 0 and 1 indicators. The value 0 is used if two lexicographic consecutive triples are equal, which means they are assigned the same lexicographic name; the value 1 increments the name in the prefix sum and assigns the unequal triple a new name.
3) Check if all lexicographic names are different by comparing the highest lexicographic name against the maximum possible (lines 12–13).
4) If all lexicographic names are different, then I_S, which contains the indices of S, is already the suffix array of the suffixes in D_3 (lines 19–20). Hence, R_1 and R_2 can be created directly: the suffix array I_S only needs to be inverted and split modulo 3. However, instead of constructing R_1 and R_2 as separate DIAs, we *interleave* them in I_R using a Sort operation such that they are balanced on the distributed system, as we will be needing pairs of mod 1/2 ranks.
5) Otherwise, prepare a recursive subproblem T_R to calculate the ranks. First, sort the lexicographic names back into string order such that $T_R = T_1 \oplus T_2$ where $\oplus$ is string concatenation (line 14). T_1 represents the complete text T using the lexicographic names of all triples $i = 1 \bmod 3$, and T_2 is another complete copy of T with triples $i = 2 \bmod 3$. By replacing the triples with lexicographic names, the original text is reduced by $\frac{2}{3}$. Second, recursively call any suffix sorting algorithm (e.g. DC3) on T_R (line 15). Last, invert the permutation SA_R to gain ranks R_1 and R_2 of triples of T in D_3, again interleave I_R such that R_1 and R_2 are distributed on the workers after the Sort operation.

With R_1 and R_2 interleaved in I_R from step D1) (lines 2–20), the objective of step D2) is to create S_0, S_1, and S_2 in line 22. Each suffix i has exactly one representative in the array S_j where $j = i \bmod 3$. Its representative contains the recursively calculated ranks of the two following suffixes in the difference cover from R_1 and R_2 (two consecutive items from I_R), and the characters $T[i], T[i+1], T[i+2], \ldots$ up to (but excluding) the next known rank.

For DC3 these are $T[i]$, $T[i+1]$, $I_R[\frac{2i}{3}]$, and $I_R[\frac{2i}{3}+1]$ for a suffix $i = 0 \bmod 3$ in S_0. $I_R[\frac{2i}{3}] = R_1[\frac{i}{3}]$ is the rank of the suffix $T[i+1..|T|)$ and $I_R[\frac{2i}{3}+1] = R_2[\frac{i}{3}]$ is the rank of suffix $T[i+2..|T|)$, which are both in the difference cover. We write the tuple as (i, c_0, c_1, r_1, r_2) where the indices are interpreted relative to $i \bmod 3$. Each suffix $i = 1 \bmod 3$ in S_1 stores $T[i]$, $R_1[\frac{i-1}{3}]$, and $R_2[\frac{i-1}{3}]$ and we write the tuples as (i, c_1, r_1, r_2) where the indices again are relative to $i \bmod 3$. And lastly, each suffix $i = 2 \bmod 3$ in S_2 stores $T[i]$, $T[i+1]$, $R_1[\frac{i-2}{3}+1]$, and $R_2[\frac{i-2}{3}]$ because $R_1[\frac{i-2}{3}+1]$ is the rank of suffix $T[i+2..|T|)$.

In the Thrill algorithm code we construct the tuples by zipping pairs from I_R, and triple groups from T together. The ZipWindow Z' (line 22) delivers $(c_0, c_1, c_2, r_1, r_2)$ for each index $i = 0 \bmod 3$. To construct the tuples in S_i two adjacent tuples need to be used because S_2's element are taken from the next tuple. This can be done in Thrill using a Window operation of size 2 (line 23). Thus to construct S_0, S_1, and S_2, we take $(c_0, c_1, c_2, r_1, r_2)$ for each index $i = 0 \bmod 3$ and $(\bar{c}_0, \bar{c}_1, \bar{c}_2, \bar{r}_1, \bar{r}_2)$ for the next index $i \bmod 3 + 3$, and output $(3i+0, c_0, c_1, r_1, r_2)$ for S_0, $(3i+1, c_0, c_1, r_1, c_2, r_2)$ for S_1, and $(3i+2, c_2, r_2, \bar{c}_0, \bar{r}_1)$ for S_2, as described above (lines 24–26). The three arrays are then sorted and merged, whereby the comparison function compares two representatives characterwise until a rank is found. The difference cover property guarantees that such a rank is found for every pair S_i, S_j during the Merge (lines 27–31).

Most of the previous discussion on DC3 can be extended to DC7 straightforwardly: Sort by seven characters instead of three, construct $T_R = T_0 \oplus T_1 \oplus T_3$ in case not all character tuples are unique, and have step D1) deliver R_0, R_1, and R_3 containing the ranks of all suffixes $(i \bmod 7) \in D_7$.

IV. Distributed External Memory Experiments

We implemented the five SACAs described in the previous sections using Thrill in C++. These implementations are available as open-source at https://github.com/thrill/thrill.

Algorithms: We prefixed all Thrill implementations in the experiment with a **T** and label prefix doubling with a Window on the inverse suffix array from Section II-A as **T.PD-Window**, prefix doubling with sorting from Section II-B as **T.PD-Sort**, and prefix doubling with discarding from Section II-C as **T.PD-Discard**. From Section III-A we include **T.DC3** and **T.DC7**. All Thrill implementations use 40-bit (5 byte) indices in the suffix array to support inputs of up to 1 TiB. They also support 48-bit and 64-bit indices.

There are only two other distributed suffix sorting implementations available. The first is pDC3 implemented using MPI by Kulla and Sanders [16]. We took their implementation, rewrote large parts, and extended it to **BKS.pDC3** and **BKS.pDC7**. Our improved pDCX version is available at https://github.com/bingmann/pDCX. These variants only support 32-bit (4 byte) indices, and thus are limited to inputs of up to 4 GiB.

The second implementation is **FA.psac** by Flick and Aluru [12] which is a highly engineered suffix sorter using MPI. It is based on prefix doubling with the inverse suffix array, but they enhanced it with alphabet compression (see Section II) in the first iteration and by using list ranking

instead of a full sort when only 1/10-th of all suffixes remain unordered. The psac implementation always works with 64-bit (8 byte) indices in the suffix array.

We also compare our distributed parallel implementations against the fastest non-distributed suffix sorters, Mori's divsufsort 2.0.2-1 [3] and sais 2.4.1 [22]. Divsufsort comes in two variants: **M.divsufsort** does not use any parallelization, and **M.divsufsort.par** uses OpenMP parallelization only in the first string sorting phase. Mori's sais is always only sequential, included as **M.sais**, and is an engineered version of SA-IS [4], [23]. We had to fix an error in sais 2.4.1 for 2 GiB inputs, and could not find a second bug which makes it crash for inputs larger than 4 GiB. We used the versions of divsufsort and sais with 64-bit (8 byte) indices.

Inputs: For our experiments we selected three inputs, which were also used by previous authors [7]. **Wikipedia** is a 125.6 GiB XML dump of the English Wikipedia dated `enwiki-201701`. **Gutenberg** is a concatenation of all text documents from Project Gutenberg by document id as available in September 2012. These total 23 GiB in size and contain a version of the human genome as a subsequence. **Pi** are the decimals of π, written as ASCII digits and starting with "3.1415." More information about the characteristics and sources of these inputs is available in the longer version of this paper [14, ch. 8]. Gutenberg and Wikipedia are diverse real-world inputs, while Pi is random. We ran some additional experiments with DNA data, but Pi delivered more consistent results for random inputs. For scalability tests, we take prefixes of size $[0 \mathinner{..} 2^k)$ of the inputs.

Platform: We ran our experiments on the Amazon Web Services (AWS) Elastic Compute Cloud (EC2) using **i3.4xlarge** instances. Each host had 16 Intel Xeon E5-2686 v4 Broadwell vCPUs with 2.30 GHz clock speed, 122 GB RAM, and 2×1.9 TB Non-Volatile Memory Express (NVMe) SSDs. We measured that these SSDs reach effective sustained sequential read speeds of 2.1 GiB/s and write speeds of 800 MiB/s each. The hosts were connected via the AWS network, and reached 1 144 MiB/s simultaneous pair-wise throughput bandwidth and 80 µs ping-pong round-trip latency in a four host test setup. All experiments were run with the Thrill master branch version from January 19th, 2018, compiled with `g++` 5.4.1 an Ubuntu Linux 16.04 "xenial" using Linux 4.4.0-1052-aws. Our suffix sorting inputs were stored on the AWS Elastic File System (EFS) and transferred via NFS to the compute hosts. In total, the experiments reported in the next section took 4 125 compute instance-hours and cost $ 1 713.

For performing distributed external memory experiments, we *limited* the available RAM on each host to 8 GB using the kernel option `mem=8G`. This leaves about 7 GB for Thrill, since the kernel reserves itself a considerable portion. This limitation is extreme, but demonstrates that Thrill can efficiently utilize disk space. For our non-distributed comparison experiments with divsufsort and sais we removed the memory limit.

A. The Results

We ran the algorithms on h instances for all powers of two ranging from 1 to 32; as each host had 16 cores, the highest core/worker count in our experiment was 512. For each host configuration, we ran the algorithms on all input prefixes from 16 MiB to $h \cdot 1$ GiB, again doubling the size in each step. Due to the quantity and considerable cost of the experiments, we only ran each configuration once. All constructed suffix arrays were verified to be correct using a checking algorithm [6, Section 8].

First consider the results shown in Figure 1. The graphs show the throughput of all suffix sorting configurations we ran in our experiment. As expected, not all algorithms succeeded in scaling to the large input sizes with limited available memory, including some of our implementations in Thrill.

The MPI implementations are clearly limited by RAM: considering that suffix sorting n bytes with 8 byte indices requires at least $9n$ bytes of RAM, at most about 770 MiB can be sorted by a single host in this setting. For 5 byte indices, the constraint rises to about 1 150 MiB on a single host. However, the MPI implementations BKS.pDC3, BKS.pDC7, and FA.psac already stopped working much earlier than these hard limits: on one and two hosts they could process at most 128 MiB, on four hosts at most 256 MiB, on eight hosts at most 512 MiB, on 16 hosts at most 1 GiB, and on 32 hosts at most 2 GiB. FA.psac performed very well on the random Pi input, which was to be expected from a prefix doubling algorithm. It was also fast on Wikipedia, but was much slower on Gutenberg. Most remarkable, however, is that it did not scale well on any input: while it was very fast on Pi for a small number of hosts, on eight or more its performance degraded quickly. On Wikipedia the suffix sorting speed did not increase well when adding more hosts. A possible reason is that the inputs and available RAM size were too small for the algorithms to reach their full potential. BKS.pDC3 and BKS.pDC7 incur the same problems as FA.psac: their performance only increases slightly with more hosts. However, their overall performance is more stable across inputs due to the underlying difference cover algorithm. While BKS.pDCX may suffer from some less well-engineered implementation details, FA.psac is high quality code, which makes its unfavorable scalability in a memory constrained environment even more surprising and unlikely.

Let us now turn to the Thrill implementations. As discussed earlier, T.PD-Window is limited in the window size by the available RAM, and hence cannot sort inputs with long LCPs. Both T.PD-Window and T.PD-Sort are slow on Gutenberg and Wikipedia, but fast on Pi, again which is to be expected from prefix doubling. T.PD-Window and T.PD-Sort are also not able to sort large prefixes of Gutenberg and Wikipedia on many hosts.

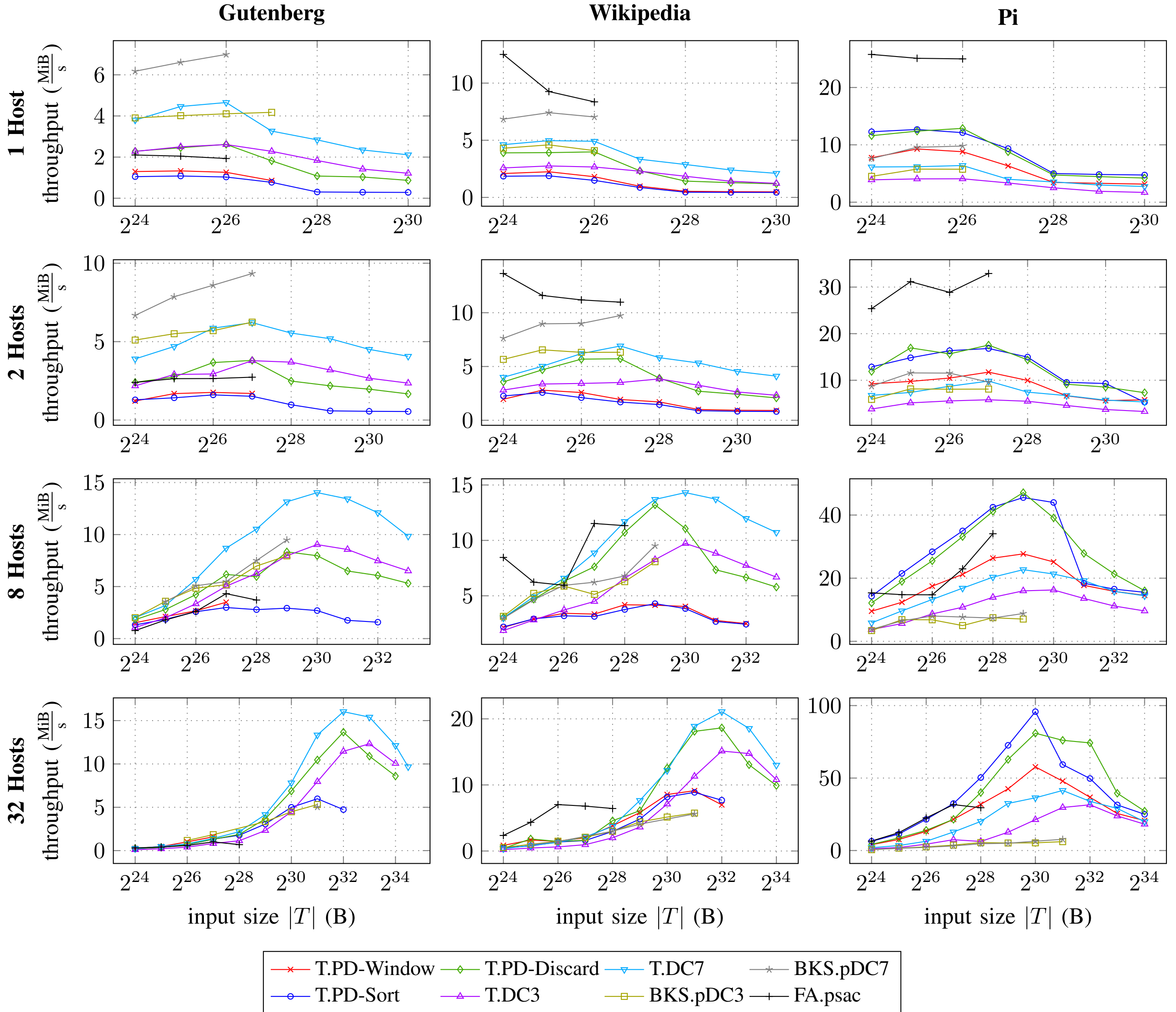

Figure 1. Throughput of distributed suffix sorters with 1–32 hosts.

The three best algorithms are T.PD-Discard, T.DC3, and T.DC7 which are able to process all inputs, except for the very largest instances. The implementations fail with 32 GiB on 32 hosts, probably because the amount of buffers and metadata in Thrill grows too large for the limited RAM available. Compared to T.PD-Window and T.PD-Sort, the discarding optimization in T.PD-Discarding really makes prefix doubling practical for larger inputs.

The throughput of all Thrill implementations increases first with the input size and also with the number of hosts, until the throughput starts dropping at 1–4 GiB size. The turning point is when external memory usage starts to impact sorting performance. This obviously must slow down the Thrill implementations, however the NVMe SSDs are so fast that the throughput only drops down to levels that many MPI implementation reach with RAM.

On the input Pi, performance peaks at 1 GiB size, reaching more than 95 MiB/s throughput with 32 hosts. For larger Pi inputs, external memory usage increases, and throughput drops to 25 MiB/s. Pi is the only input for which our prefix doubling T.PD-Sort works well; the more complex T.PD-Discard is second best and gains no advantage over regular prefix doubling algorithms for this uniformly random input. On Wikipedia and Gutenberg the picture changes completely: unoptimized prefix doubling algorithms are much slower and often even fail to sort the input. Clearly the best

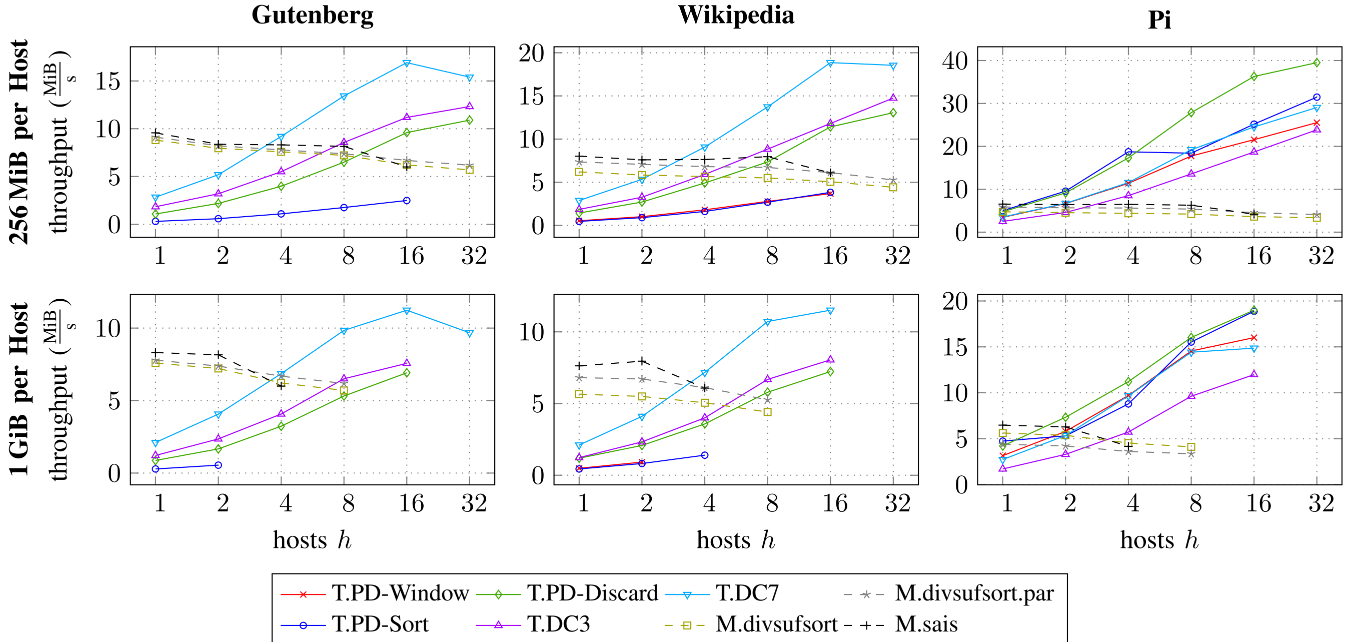

Figure 2. Weak scaling plots of distributed and of the fastest non-distributed suffix sorters run on one host with the same input size.

implementation for these inputs is T.DC7, which outperforms on nearly all instance of Wikipedia and Gutenberg. T.DC3 appears to be slower by a constant factor, and T.PD-Discard also performs very well, but with a different characteristic than the difference cover algorithms.

To better compare the scalability properties, we present weak scaling plots in Figure 2. These can be considered *slices* of the previous diagrams: the left panels shows all experiments with $h \cdot 256$ MiB input per host and the right panels all with $h \cdot 1$ GiB per host. Only Thrill implementations function properly with these parameters. Additionally, we include results from the non-distributed algorithms M.divsufsort, M.divsufsort.par, and M.sais. These run on only *one* host, but with the *same* amount of input data as the distributed experiments. As these three implementations are the fastest non-distributed suffix sorters, we can determine the number of hosts needed for distributed algorithms to outperform.

Despite the full 122 GB RAM available, M.divsufsort, M.divsufsort.par, and M.sais could not suffix sort 16 GiB inputs or larger, because the algorithms require at least $9n$ working space. M.sais failed even for 8 GiB due to an unknown error in the program. T.DC7 outperformed M.divsufsort on Gutenberg and Wikipedia with four hosts, and on Pi T.PD-Discard outperformed already for two host. Considering these numbers, one has to bear in mind, that Thrill uses all 16 cores on the hosts, while M.divsufsort only uses one. M.divsufsort.par uses OpenMP parallelism, but that does not have a large impact. M.sais is slightly faster than M.divsufsort on our inputs. Previous experiments on the performance of big data frameworks [24] reported "Configuration that Outperforms a Single Thread (COST)" ratios of 16 to 512 for PageRank, and 10 to 100 for graph connectivity. The COST ratio of our suffix sorters are thus 32–64. However, we were unable to replicate the 110x speedup reported for FA.psac [12] over M.divsufsort, probably due to our cheaper commodity hardware.

V. Conclusions and Future Work

We presented the implementation of five different suffix array construction algorithms in Thrill showing that the small set of algorithmic primitives provided by Thrill is sufficient to express the algorithms within the framework. Our experimental results demonstrate that algorithms implemented in Thrill are competitive to hand-coded MPI implementations. Furthermore, Thrill already has automatic external memory support, hence our implementations are the first distributed external memory suffix array construction algorithms.

While our experimental results are already impressive, we believe that more future work should be directed at improving efficiency of the underlying sorting algorithm implementations in Thrill. The suffix sorting algorithms are the most complex algorithms currently implemented in the framework, and by improving their performance, all other applications will also gain. In addition, one could extend the SACAs with LCP array construction and the difference cover algorithms with discarding tuples [25] similar to the technique we applied to the prefix doubling algorithms.

And then one can turn to post-processing the suffix array into other forms such as compressed indices, the FM-index, or specific on-disk suffix array representations like RoSA [26].

Implementing these efficiently and scalable using the Thrill framework will open up new possibilities for applying advanced text algorithms to large datasets.

Acknowledgment

We would like thank the AWS Cloud Credits for Research program for making the experiments in Section IV possible. Our research was supported by the Gottfried Wilhelm Leibniz Prize 2012, the German Research Foundation (DFG) SPP 1736 priority programme "Algorithms for Big Data", and the Large-Scale Data Management and Analysis (LSDMA) project in the Helmholtz Association.

References

[1] U. Manber and G. Myers, "Suffix arrays: A new method for on-line string searches," *SIAM Journal on Computing*, vol. 22, no. 5, pp. 935–948, Oct. 1993.

[2] G. H. Gonnet, R. A. Baeza-Yates, and T. Snider, *New Indices for Text: PAT Trees and PAT Arrays*. Prentice-Hall, Jun. 1992, ch. 5, pp. 66–82.

[3] Y. Mori, "DivSufSort," 2006, first version. [Online]. Available: https://github.com/y-256/libdivsufsort

[4] G. Nong, S. Zhang, and W. H. Chan, "Linear suffix array construction by almost pure induced-sorting," in *Data Compression Conference (DCC)*. IEEE, Mar. 2009, pp. 193–202.

[5] A. Crauser and P. Ferragina, "A theoretical and experimental study on the construction of suffix arrays in external memory," *Algorithmica*, vol. 32, no. 1, pp. 1–35, 2002.

[6] R. Dementiev, J. Kärkkäinen, J. Mehnert, and P. Sanders, "Better external memory suffix array construction," *ACM Journal of Experimental Algorithmics (JEA)*, vol. 12, pp. 3.4:1–24, Aug. 2008.

[7] T. Bingmann, J. Fischer, and V. Osipov, "Inducing suffix and LCP arrays in external memory," in *15th Workshop on Algorithm Engineering & Experiments (ALENEX)*. SIAM, Jan. 2013, pp. 88–102.

[8] J. Kärkkäinen, D. Kempa, and S. J. Puglisi, "Parallel external memory suffix sorting," in *26th Symposium on Combinatorial Pattern Matching (CPM)*, ser. LNCS, vol. 9133. Springer, Jun. 2015, pp. 329–342.

[9] T. Bingmann, M. Axtmann, E. Jöbstl, S. Lamm, H. C. Nguyen, A. Noe, S. Schlag, M. Stumpp, T. Sturm, and P. Sanders, "Thrill: High-performance algorithmic distributed batch data processing with C++," in *IEEE International Conference on Big Data*. IEEE, Dec. 2016, pp. 172–183.

[10] S. J. Puglisi, W. F. Smyth, and A. Turpin, "A taxonomy of suffix array construction algorithms," *ACM Computing Surveys*, vol. 39, no. 2, Jul. 2007.

[11] J. Labeit, J. Shun, and G. E. Blelloch, "Parallel lightweight wavelet tree, suffix array and FM-index construction," *Journal of Discrete Algorithms (JDA)*, vol. 43, pp. 2–17, Mar. 2017.

[12] P. Flick and S. Aluru, "Parallel distributed memory construction of suffix and longest common prefix arrays," in *28th International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. ACM, Nov. 2015, pp. 16:1–16:10.

[13] J. Kärkkäinen, P. Sanders, and S. Burkhardt, "Linear work suffix array construction," *Journal of the ACM (JACM)*, vol. 53, no. 6, pp. 918–936, Nov. 2006.

[14] T. Bingmann, "Scalable string and suffix sorting: Algorithms, techniques, and tools," Ph.D. dissertation, Karlsruhe Institute of Technology, Germany, Jul. 2018, doi: 10.5445/IR/1000085031, ISBN: 978-1727532128.

[15] J. Kärkkäinen and P. Sanders, "Simple linear work suffix array construction," in *International Colloquium on Automata, Languages, and Programming (ICALP)*, ser. LNCS, vol. 2719. Springer, Jun. 2003, pp. 943–955.

[16] F. Kulla and P. Sanders, "Scalable parallel suffix array construction," *Parallel Computing*, vol. 33, no. 9, pp. 605–612, 2007.

[17] T. Bingmann, J. Fischer, and V. Osipov, "Inducing suffix and LCP arrays in external memory," *ACM Journal of Experimental Algorithmics (JEA)*, vol. 21, no. 2, pp. 2.3:1–27, Sep. 2016.

[18] J. Kärkkäinen, D. Kempa, S. J. Puglisi, and B. Zhukova, "Engineering external memory induced suffix sorting," in *19th Workshop on Algorithm Engineering & Experiments (ALENEX)*. SIAM, Jan. 2017, pp. 98–108.

[19] N. J. Larsson and K. Sadakane, "Faster suffix sorting," *Theoretical Computer Science (TCS)*, vol. 387, no. 3, pp. 258–272, 2007.

[20] D. Weese, "Entwurf und implementierung eines generischen substring-index," May 2006, diploma Thesis. Humboldt University Berlin, Germany, in German. [Online]. Available: http://publications.imp.fu-berlin.de/457/

[21] J. Singer, "A theorem in finite projective geometry and some applications to number theory," *Transactions of the American Mathematical Society*, vol. 43, no. 3, pp. 377–385, 1938.

[22] Y. Mori, "SAIS: An implementation of the induced sorting algorithm," 2008. [Online]. Available: https://sites.google.com/site/yuta256/sais

[23] G. Nong, S. Zhang, and W. H. Chan, "Two efficient algorithms for linear time suffix array construction," *IEEE Transactions on Computers*, vol. 60, no. 10, pp. 1471–1484, Sep. 2011.

[24] F. McSherry, M. Isard, and D. G. Murray, "Scalability! but at what COST?" in *15th Workshop on Hot Topics in Operating Systems (HotOS)*. USENIX Association, May 2015.

[25] S. J. Puglisi, W. F. Smyth, and A. Turpin, "The performance of linear time suffix sorting algorithms," in *Data Compression Conference (DCC)*. IEEE, Mar. 2005, pp. 358–367.

[26] S. Gog and A. Moffat, "Adding compression and blended search to a compact two-level suffix array," in *20th International Conference on String Processing and Information Retrieval (SPIRE)*, ser. LNCS, vol. 8214. Springer, Oct. 2013, pp. 141–152.