Engineering a Distributed Full-Text Index*

Johannes Fischer† Florian Kurpicz† Peter Sanders‡

Abstract

We present a distributed full-text index for big data applications in a distributed environment. Our index can answer different types of pattern matching queries (existential, counting and enumeration). We perform experiments on inputs up to 100 GiB using up to 512 processors, and compare our index with the distributed suffix array by Arroyuelo et al. [Parall. Comput. 40(9): 471–495, 2014]. The result is that our index answers counting queries up to 5.5 times faster than the distributed suffix array, while using about the same space. We also provide a succinct variant of our index that uses only one third of the memory compared with our non-succinct variant, at the expense of only 20% slower query times.

1 Introduction

Index data structures are one of the most powerful tools for coping with large data sets. Among the index data structures for texts are suffix arrays, suffix trees and related structures that allow full-text search of patterns in time *independent* of the size of the text corpus. Consequently, there are thousands of papers on such suffix data structures. However, when you look at truly large data sets that do not fit on a single machine, there is very little work yet. We found that very surprising since the biggest inputs are those where the index has the biggest impact. Refer to § 1.2 for a more detailed discussion of related work. You can easily adopt the approach from commercial search engines (which use inverted index data structures) to distribute your corpus over the machines and then use a local index on each machine. However, then the amount of work and energy you invest in a query grows proportional to the number of machines and thus, *linearly* with the corpus size – you have (asymptotically) thrown away the huge advantage of a powerful index data structure.

Our contribution is the development of a truly distributed full-text index data structure that supports typical queries by exchanging only a constant number of messages whose length is proportional to the length of the search pattern.

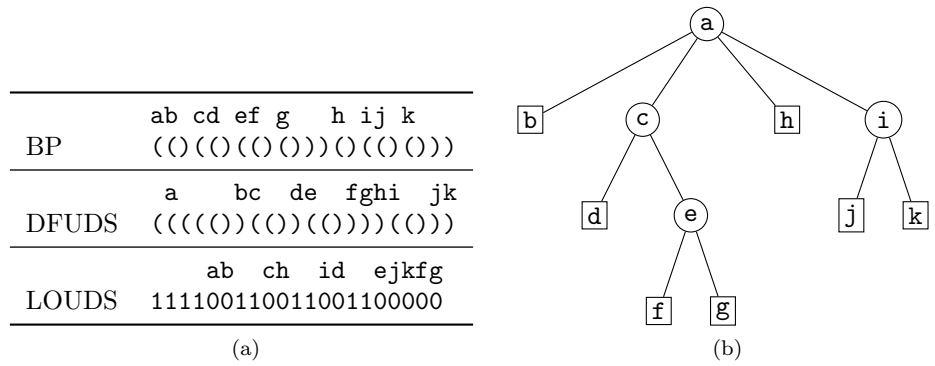
1.1 Overview. Our construction algorithm starts from a distributed input text T and a distributed suffix array (SA) together with information on the longest common prefix of subsequent entries in the SA, the so called LCP-array. In § 2 we formally introduce the SA, LCP-array, and other prerequisites. By scanning the SA and LCP-array, we then construct a two-level trie data structure. Using succinct data structures we can reduce the size of the trie to 15 bits per character of the text. In our scenario, a *processing elements* (PE) denotes a processor on a compute node in the cluster. A small top-level trie GT (see § 3.1 for more details) is sufficient to decide which PEs are involved in answering a query. GT is replicated over all PEs so that queries can arrive anywhere and get forwarded to those PEs that hold the relevant part of the SA. The search in the local part of a SA is facilitated by a succinctly represented Patricia trie on that part. After this local search, a single remote access to the text suffices to locate the pattern in the SA. Refer to § 3.1 for a more detailed explanation of our data structure.

1.2 Related Work. Multi-level full-text indices have been considered for external memory. The *String B-Tree* [6] utilizes Patricia tries at each level to reduce the I/O volume. There exists also theoretical work by Ferragina and Luccio [7] that discusses a distributed Patricia trie. Their approach is only good when answering *long* queries, for example, existential queries "does the pattern occur in the text?" of length $m \geq c$, where c is the number of PEs. Those queries can be answered optimally with respect to computation and communication.

*This work was supported by the German Research Foundation (DFG), priority programme "Algorithms for Big Data" (SPP 1736).

†Technische Universität Dortmund, Department of Computer Science, johannes.fischer@cs.tu-dortmund.de, florian.kurpicz@tu-dortmund.de

‡Karlsruhe Institute of Technology, Institute for Theoretical Informatics, sanders@kit.edu



(a) (b)

Figure 1: (a) Succinct representations of the tree shown in (b).

For real world applications this does not provide a satisfactory solution, as the average length of natural queries is shorter (18 AOL [27] and 21 TREC [26] on average) than the number of PEs we want to utilize.

A complementary (theoretical) approach is described by Mäkinen et al. [21] and is good for short patterns. Using backwards search, a query can be answered using m communication steps. The problem is that at most σ PEs can be used and, in the worst case, some PEs might need space $\Omega(n)$.

Arroyuelo et al. [1] consider a large variety of distributed suffix array data structures with various tradeoffs between replication, number of remote data accesses, and load balance. However, they always use explicit binary search in the suffix array, leading to logarithmically higher costs than our approach. Our index can be viewed as an improvement of the *global* approach presented in [1], where we add two levels of tries that allow us to answer each query exchanging only a constant number of messages. We show that our index scales better especially for larger text sizes and a larger number of PEs.

Russo et al. [28] (theoretically) describe distributed compressed indices. Their approach partitions the text between the PEs and works with local indices. The consequence is that queries have to be processed on every PE, contrary to our goal to have total work independent of c.

It is known that the suffix array can be computed efficiently in parallel [17, 18]. Flick et al. [11] give the only distributed algorithm for computing both the SA and LCP-array; their approach is within a factor of $\mathcal{O}(\log n)$ from the optimal. We use the SA and LCP-array as the starting point for our index construction and we do not study their construction in this paper. Note that none of these papers [11, 17, 18] discuss how to actually use the resulting data structure for a distributed query.

There are several results that assume that the input text is replicated over all PEs (e.g. [5]). This makes index construction and search much easier but severely limits scalability, so we do not consider them further.

2 Preliminaries

2.1 Suffix Arrays. Let $\mathsf{T} = \mathsf{T}[1]\,\mathsf{T}[2]\ldots\mathsf{T}[n]$ be a text of length n over an alphabet Σ of size $|\Sigma| = \sigma$. $\mathsf{T}[i..j]$ denotes the *substring* $\mathsf{T}[i]\ldots\mathsf{T}[j]$ for all $1 \le i \le j \le n$. Also, $P_i = \mathsf{T}[1..i]$ is i-th *prefix* of T and $S_i = \mathsf{T}[i..n]$ is the i-th *suffix* of T for all $1 \le i \le n$. The longest common prefix (lcp) denotes the maximum size of a common prefix of two suffixes of the text, i.e., $\mathsf{lcp}(i,j) = \max\{k \ge 0 : \mathsf{T}[i..i+k-1] = \mathsf{T}[j..j+k-1]\}$ for all $1 \le i, j \le n$.

The *suffix array* [22] SA of a text T with $|\mathsf{T}| = n$ is a permutation of $[1, n]$ such that it enumerates the starting positions of the suffixes of T in lexicographical order. Hence, $S_{\mathsf{SA}[i]} <_{\text{lex}} S_{\mathsf{SA}[j]}$ for all $1 \le i < j \le n$. The *longest common prefix array* (LCP-array) enhances the SA and contains the size of the longest common prefix of two lexicographically consecutive suffixes, i.e., $\mathsf{LCP}[1] = 0$ and $\mathsf{LCP}[i] = \mathsf{lcp}(\mathsf{SA}[i-1], \mathsf{SA}[i])$ for all $2 \le i \le n$.

The SA and LCP-array are the foundation of the indices considered in this paper, i.e., the index we present and the index we compare our results with. Not only can they be simulated using these two arrays, but they can also be constructed using only the SA, LCP-array and T. As already mentioned in the introduction, there exists a distributed algorithm for constructing the SA and LCP-array [11], which is a distributed variant of Larsson and Sadakane's algorithm [19]. Using c PEs it can compute the SA in $\mathcal{O}(t_{\text{sort}}(n,c)\lg n)$ time, where $t_{\text{sort}}(n,c)$ denotes the time required to sort n elements that are uniformly distributed over c PEs. The LCP-array construction

requires $\mathcal{O}\big(\big(\frac{n}{c}+c\big)\lg n\big)$ additional time. Another distributed SA construction algorithm is the pDC3 [18], which is a distributed variant of the DC3 algorithm [17], can compute the SA in time $\mathcal{O}\Big(\frac{n\lg n}{c}+\lg^2 c\Big)$, i.e., a log-factor better than [11]. For our experiments we use the implementation of the latter, as it is the only available distributed SA and LCP-array construction algorithm.

2.2 Tries. Given a labeled tree $G=\langle V,E\rangle$ with root $r\in V$, we denote the label of a node or an edge by $\mathsf{label}(\cdot)$ and the concatenation of all edge labels on the path from the root to a node v by $\mathsf{pathlabel}(v)$. The *out-degree* of a node v is denoted by $\delta^+(v)$. The *leaf rank* of a leaf $\ell\in V$ is the number of leaves visited before ℓ in a preorder traversal of the tree.

Let $R=\{R_1,R_2,\ldots,R_k\}$ be a set of strings over the alphabet Σ such that all strings are distinct and no string is the prefix of another string in R. The *trie* of R is an ordered tree with root r, where the edge labels are characters and the leaves represent string numbers from $[1,k]$ such that:

1. for each node $v\in V$, the labels of the outgoing edges $\mathsf{label}((v,\cdot))\in\Sigma$ are distinct,
2. for each string $R_i\in R$, there is a leaf $\ell\in V$ with $R_i=\mathsf{pathlabel}(\ell)$ and $\mathsf{label}(\ell)=i$ and
3. for each leaf $\ell\in V$ there is a string $R_i\in R$ such that $R_i=\mathsf{pathlabel}(\ell)$ and $\mathsf{label}(\ell)=i$.

The *compressed trie* is a trie where each path $e_1,e_2,\ldots,e_\ell$ with $\ell>1$ consisting only of nodes with out-degree 1 is replaced by a single edge e such that $\mathsf{label}(e)=\mathsf{label}(e_1)\,\mathsf{label}(e_2)\ldots\mathsf{label}(e_\ell)$. Still, all outgoing edges of a node v start with a different character. The *string depth* of a node v is $\mathsf{sd}(v)=|\mathsf{pathlabel}(v)|$, i.e., the length of the longest common prefix of all strings represented leaves below v. To find all occurrences of a pattern P in a compressed trie, we start at the root r and follow the edge e such that $\mathsf{label}(e)=\mathsf{P}[1..|\mathsf{label}(e)|]$. At each node v, the length of the pattern matched up to this point equals $\mathsf{sd}(v)$. We then follow the edge e with $\mathsf{label}(e)=\mathsf{P}[\mathsf{sd}(v)+1..|\mathsf{label}(e)|+\mathsf{sd}(v)]$. This process is repeated until we have matched the whole pattern at the edge $(\cdot,v)$. Then, all leaves that are successors of v correspond to strings in R that are prefixed by P. If at any point, there is no edge to follow, the pattern P does not occur in the trie.

The *Patricia trie* (or *blind trie*) [23] of a text T is a compressed trie for all suffixes of T, where each node v just stores the first character and the string depth $\mathsf{sd}(v)$. Due to this limitation, finding all occurrences of a pattern requires two steps – a *blind search* followed by a comparison to a substring of T (which has been determined by the blind search):

For the blind search, we start at the root and follow the edge matching the pattern at the position corresponding to the string depth, i.e., at a node v we follow the edge e with label $\mathsf{label}(e)=\mathsf{P}[\mathsf{sd}(v)]$. We repeat this until we have reached a node v such that $\mathsf{sd}(v)\geq|\mathsf{P}|$ or there is no feasible edge to follow. In the first case, we retrieve a prefix of length $|\mathsf{P}|$ of a suffix corresponding to any leaf w that is a successor of v and compare that prefix with our pattern P. In the second case (there is no edge to follow) P does not occur in T.

Next, we compare P and $\mathsf{T}[i..i+|\mathsf{P}|-1]$ where i is the label of the leaf w (that has been identified during the blind search). If the strings are equal, then all leaves that are below v correspond to an occurrence of P in T. Otherwise, P does not occur in T.

The Patricia trie can be constructed from the SA, LCP-array and T in linear time, i.e., scanning the SA and LCP-array once and considering each entry at most twice. T is required for the edge labels and each position is accessed at most once. In § 3.1 we give a detailed description of the construction algorithm and in § 4.1 we compare the construction time for the tries needed by our index with the time required for the construction of the SA and LCP-array.

2.3 Succinct Data Structures. We can represent a tree containing ℓ nodes using a bit vector $B\in\{0,1\}^{2\ell}$. The bits represent parentheses; a 1 represents an open parenthesis "(" and a 0 represents a closing parenthesis ")". To navigate in the tree, we require additional operations on the bit vector: $\mathsf{rank}_0(i)$ asks for the number of 0's in B up to position $i-1$, $\mathsf{select}_0(i)$ returns the position of the i-th 0 in B, and $\mathsf{find\_close}(i)$ gives the position of the matching closing parenthesis for an open parenthesis at position i. The operations $\mathsf{rank}_1(\cdot)$ and $\mathsf{select}_1(\cdot)$ work analogously for 1's in the bit vector. All these operations can be answered in constant time [4, 15]. The *level ordered unary degree sequence* (LOUDS) [15] represents a tree level-wise, i.e., starting at the root, we visit all nodes v of a level from left to right and add $\delta^+(v)$ 1's followed by a 0 to the bit vector. The position of the i-th

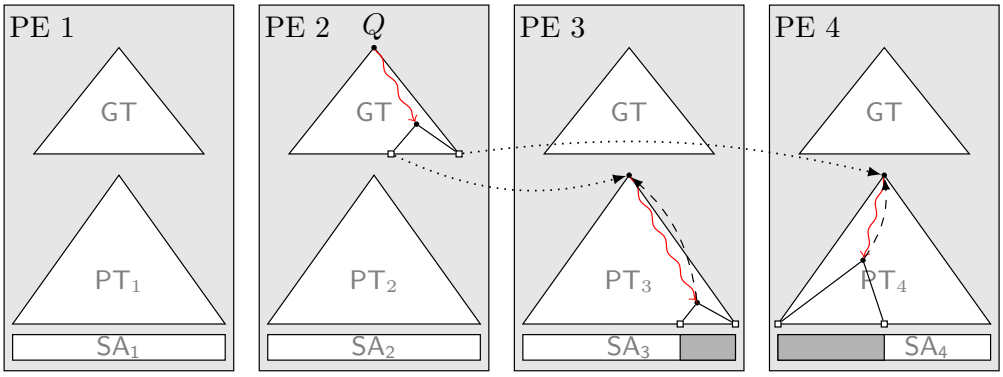


Figure 2: The first level of the DPT is a trie (GT) over the smallest and largest suffix stored by the local suffix array at each PE. It is the same at each PE. The lower level at PE i is the local Patricia trie PT_i over SA_i. A query Q is answered in four supersteps. First, the PEs that are responsible for the query are determined (using the trie GT at the first level). Then the query is sent to those PEs. In the second superstep, a blind search in the Patricia trie PT_i (second level of PE i) is executed. The substring corresponding to the result of the blind search is retrieved and in the third superstep the query is answered using that substring.

child of the node at position x is identified by $\mathsf{select}_0(\mathsf{rank}_1(x)+i-1)+1$ in constant time. The *depth first unary degree sequence* (DFUDS) [2] is obtained by traversing the tree in preorder and (like in LOUDS) append $\delta^+(v)$ 1's followed by a 0 whenever we visit a node v for the first time. To make the sequence balanced, we prepend a 1. The position of the i-th child of the node at position x is identified by $\mathsf{find\_close}(\mathsf{select}_0(\mathsf{rank}_1(x)+1)+1)+1$ in constant time. Last, the *balanced parenthesis* (BP) representation [24] is also constructed by traversing the tree in preorder: We add a 1 to the bit vector whenever we visit a node for the first time and we add a 0 to the bit vector whenever we visit a node for the last time. In theory, BP also allows an access of the i-th child in constant time [29]. However, in the implementation used by us (see § 4 for more details), BP does not support a direct access to the i-th child. Instead, one has to access the first child of the node at position x (position $x+1$) and then go to the next child ($\mathsf{find\_close}(x)+1$) until the i-th child is reached in $\mathcal{O}(i)$ time. In our implementation, the first two representations allow an access of the i-th child in in constant time, whereas it takes $\mathcal{O}(i)$ time in BP. See Figure 1 for an example of these succinct tree representations.

2.4 Model of Computation. In the *bulk-synchronous parallel* (BSP) model [31], each computation is a sequence of *supersteps*. Each superstep is split into three parts. First, the PEs can perform any number of operations based on local data. This is followed by a communication phase, where the PEs can send data to other PEs. After the communication, all PEs are synchronized, i.e., all PEs wait until the last PE has finished all operations on local data and communication. There is no synchronization between the first and second part, PEs can start communicating as soon as they have finished working on the local data (the results of the communication are not available during the superstep). Then at the beginning of the next superstep, each PE can use the data retrieved during the last superstep. The total running time of a BSP program is the cost of all its supersteps, where the cost of one superstep is $w+hG+L$. Here,

- w is the maximum time used for computation by each PE (excluding communication),
- h is the maximum of machine words communicated by each PE, with G being the running time required for the communication of one machine word, and
- L is the cost of the barrier synchronization.

BSP has been considered as the model for other distributed indices [1, 7]. Also, we will see that the BSP model suits our setting well, as we have well defined phases where communication is required during construction and query processing – see § 3. Whenever we *retrieve* data we refer to *direct remote memory access* (DRMA) that is supported by the BSP model [30].

3 Distributed Patricia Trie

Let $\mathsf{T} = \mathsf{T}[1]\,\mathsf{T}[2]\ldots\mathsf{T}[n]$ be a text of length n. The PEs are numbered from 1 to c. We assume that n is divisible by c. Then we distribute the SA and LCP-array in a consecutive fashion, such that the i-th PE holds $\mathsf{SA}_i = \mathsf{SA}[1 + (i-1)\,\frac{n}{c}..i\frac{n}{c}]$ and $\mathsf{LCP}_i = \mathsf{LCP}[1 + (i-1)\,\frac{n}{c}..i\frac{n}{c}]$. In addition, each PE holds a part of the text as described in the next section.

Our proposed data structure, the *distributed Patricia trie* (DPT), is a two level index consisting of an index GT for query distribution (first level) and several indices PT_i that can find all occurrences of a pattern P that starts at text positions held by the local SA_i on PE i (second level). In this case we say that the PE i is *responsible* for P. The index GT is replicated at every PE. This allows queries to arrive at arbitrary PEs and then to be sent to the responsible PEs in the next step. There, the query is processed utilizing PT_i. This index is unique for each PE – see also Figure 2.

3.1 Construction. In this section we show how to construct the DPT in linear time. We start with the construction of the local PTs as we use the information about their smallest and greatest element for the construction of GT.

Local Tries. The construction is the same at each PE. Our construction algorithm is the extension of an algorithm to compute the *suffix tree*, i.e., a compressed trie of all suffixes of a text T. We modified the suffix insertion algorithm presented in [20, p. 143] such that the Patricia trie can be constructed by scanning the SA and LCP-array from left to right. The pathlabel of the rightmost path in a (Patricia) trie is the lexicographically largest pathlabel in the trie. Since all suffixes in SA are in lexicographical order, each suffix that is added to the Patricia trie is lexicographical greater that all previously inserted suffixes and will form the new rightmost path. Therefore, at each point of time during the construction, only nodes on the rightmost path can be changed. All other nodes are considered as *final*. The inner nodes on the rightmost path, i.e., the nodes that can still be changed, are kept on a stack. Since we compute a Patricia trie, each node v knows its string depth $\mathsf{sd}(v)$.

Initially, we have a stack containing a node with string depth 0 and no children. We start by adding the first inner node v, with $\mathsf{sd}(v) = \mathsf{LCP}[2]$, two children (the left child represents $\mathsf{SA}[1]$ and the right child represents $\mathsf{SA}[2]$) with edge labels $\mathsf{T}[\mathsf{SA}[1] + \mathsf{LCP}[2]]$ and $\mathsf{T}[\mathsf{SA}[2] + \mathsf{LCP}[2]]$, resp. If $\mathsf{sd}(v) = 0$, the node replaces the initial one that has been on the stack. Otherwise, v will be a child of the initially created node. We now continue to scan the SA and LCP-array from left to right. Whenever we read a new position i in the LCP-array, we remove nodes from the stack until the node v on top of the stack has $\mathsf{sd}(v) \leq \mathsf{LCP}[i]$. If $\mathsf{sd}(v) < \mathsf{LCP}[i]$, we create a new inner node w with $\mathsf{sd}(w) = \mathsf{LCP}[i]$, i.e., we *branch below* node v. The left child of w (edge label $\mathsf{T}[\mathsf{SA}[i-1] + \mathsf{LCP}[i]]$) is the former rightmost child of v, and the right child of w is a new leaf referring to $\mathsf{SA}[i]$ and has edge label $\mathsf{T}[\mathsf{SA}[i] + \mathsf{LCP}[i]]$. Next, w becomes the new rightmost child of v and is put on the stack. If $\mathsf{sd}(v) = \mathsf{LCP}[i]$, v just gets a new rightmost child (edge label $\mathsf{T}[\mathsf{SA}[i] + \mathsf{LCP}[i]]$) referring to $\mathsf{SA}[i]$, i.e., v gets a new *leaf*. Following these operations, we can compute each local PT in $\mathcal{O}\big(\frac{n}{c}\big)$ time.

With respect to practical application, we also want to construct succinct representations of the tries. It is possible to compute a succinct representation using its pointer based representation. Using the approach described above, we can also compute a succinct trie representation directly, i.e., reducing the required memory peak for the construction. We compute the DFUDS representation of a trie by storing all final nodes and their subtrees in DFUDS representation. Whenever we remove a node from the stack, we add it and its subtree at the end of the already computed DFUDS representation of the previously removed final nodes. This is possible because we construct the trie in the same order as a depth first search traversal visits all nodes (which is the order in which the nodes are represented in DFUDS).

Up to now, we have simply named the characters that correspond to the edge labels. Since all local PTs are on different PEs, we cannot assure that the text position required for an edge label is locally available. We have to retrieve all edge labels during one communication phase. The number of characters stored at each PE is $\Theta(\frac{n}{c})$. During the construction of the local Patricia trie PT_i, we scan the arrays SA_i and LCP_i to determine the first mismatching text positions of two lexicographically consecutive suffixes. These characters will then be used for the edge labels later on. If we create a new leaf, we only require *one* edge label.

For a simpler and more realistic analysis of the costs for constructing our local indexes, we assume that all mismatching characters are stored at the same PE where the corresponding suffix starts, i.e., we assume that $\mathsf{T}[\mathsf{SA}[i]..\,\mathsf{SA}[i] + \max(\mathsf{LCP}[i], \mathsf{LCP}[i+1])]$ is stored on one PE for all $1 \leq i \leq n$. This is usually the case if the text T is composed of a number of smaller documents such that all documents reside on a *single* PE, but all PEs still

have $\Theta\left(\frac{n}{c}\right)$ characters. (If this is not the case, one could still replicate parts of the text on each PE such that the PEs hold overlapping parts of the text.) Under this assumption, each PE needs to *send* $\mathcal{O}\left(\frac{n}{c}\right)$ characters as edge labels. Further, each PE also *receives* at most $\mathcal{O}\left(\frac{n}{c}\right)$ characters, as the local Patricia trie has less than $2\frac{n}{c}$ edges. Finally, we note that the construction takes one superstep (construct the tree and store the text positions, then retrieve the characters at those positions). This leads to the following Lemma (where the claim about space follows easily because the local Patricia tries store the edge labels, the SA-values at the leaves, and the skip values at the edges).

LEMMA 3.1. *Given that all mismatching characters are stored at the same PE where the corresponding suffix starts, the* SA*, and the* LCP*-array, constructing the Patricia tries costs* $\mathcal{O}\left(\frac{n}{c} + \frac{n}{c}G + L\right)$*. Each* PT_i *requires* $|\textit{tree structure}| + \mathcal{O}\left(\frac{n}{c}\left(\lg n + \lg \sigma\right)\right)$ *bits of space.*

Global Trie. Next, we consider the construction of the *global trie* GT, which allows us to distribute queries without accessing the text. GT is the same at every PE, which allows arbitrary PEs to initially process any query. To identify all PEs that are responsible for a pattern, we require the smallest and largest suffix that is represented by each PE and their lcp-values. Using this set of suffixes $\mathcal{S} = \{S_{\mathsf{SA}_1[1]}, S_{\mathsf{SA}_1[\frac{n}{c}]}, \ldots, S_{\mathsf{SA}_c[1]}, S_{\mathsf{SA}_c[\frac{n}{c}]}\}$ to construct GT, we can use the following observation to identify all PEs that are responsible for a pattern.

OBSERVATION 3.2. *PE i is responsible for a pattern* P *if and only if* $\mathsf{T}[\mathsf{SA}_i[1]..\,\mathsf{SA}_i[1] + |\mathsf{P}| - 1] \leq_{lex} \mathsf{P}$ *and* $\mathsf{P} \leq_{lex} \mathsf{T}[\mathsf{SA}_i[\frac{n}{c}]..\,\mathsf{SA}_i[\frac{n}{c}] + |\mathsf{P}| - 1]$.

Obviously, there can be pattern for which multiple PEs are responsible. Depending on the type of query, we need to use the second level index of at most two PEs to answer a query. Communication with more PEs may be necessary – see § 3.2 for more details.

The global trie can be constructed similar to the local Patricia trie construction described above. The suffixes required for the construction are known (all suffixes in $\mathcal{S}$). We still require the size of the longest common prefixes of those suffixes. For two lexicographically consecutive suffixes the size is in the LCP-array. The size of the longest common prefix of the other suffixes is the string depth of the root of the corresponding local PT. We can propagate all these values during one communication phase, where each PE sends the two text positions and lcp-values to all other nodes, sending $\mathcal{O}(c)$ messages of constant size. At the end of the phase each PE has a temporary SA and a temporary LCP-array each of size $2c$. Using these arrays we use the algorithm described above, only handling edge labels differently.

The task of the global trie GT is to distinguish all elements in $\mathcal{S}$ without accessing T. Therefore, the edge labels may consist of more than one character. The first character of an edge (v, w) is the same character we would store if we constructed a Patricia trie. Let the text position of this character be i. Instead of storing only this character and the string depth, we now need to store the substring $\mathsf{T}[i..i + \mathsf{sd}(w)]$ as the edge label of (v, w). Hence, it is not necessary to store the string depth at the nodes. In addition, we construct the trie with respect to a maximum pattern size $|\mathsf{P}|_{\max}$. We can usually assume that $|\mathsf{P}|_{\max}$ is constant (chosen during construction) such that the size $|\mathsf{P}|$ of each pattern P is at most $|\mathsf{P}|_{\max}$. Thus, the total size of all edge labels is bounded by $2c\,|\mathsf{P}|_{\max}$. During the construction, we store references to the edge labels, i.e., the text position and length. Therefore, at each PE it is known which substrings need to be communicated (as edge labels). The edge labels are distributed among the PEs in two supersteps. First, each PE sends an equal amount of different labels to each PE. The cost for this superstep (including the construction of the tree structure) is $\mathcal{O}(c + c\,|\mathsf{P}|_{\max} G + L)$. In the next superstep, each PE distributes the received labels to each other PE, costing $\mathcal{O}(c\,|\mathsf{P}|_{\max} G + L)$.

To prevent that a requested substring spans over more than one PE we pad the locally stored text with the next $|\mathsf{P}|_{\max}$ characters. Since we build the trie for $2c$ substrings, this requires $\mathcal{O}(c)$ time, which leads to the following Lemma.

LEMMA 3.3. *Given the* SA *and* LCP*-array, constructing the global trie costs* $\mathcal{O}(c + c\,|\mathsf{P}|_{\max} G + L)$*. The trie requires* $|\textit{tree structure}| + \mathcal{O}(c\,|\mathsf{P}|_{\max} \lg \sigma)$ *bits of space.*

Reducing the Memory Overhead. Now we show how we can reduce the memory overhead by increasing the number of supersteps required during construction. The whole index is kept in main memory, therefore, we want the overhead during the construction to be as small as possible. First, note that we can stream the SA

and LCP-array, since we just need to scan them once for the construction. Second, we look at the size of the indices, as usually a text position requires more space than a character. Since we need characters but obtain text positions, we need to store them until the next communication phase. Usually $\lg n \gg \lg \sigma$ (i.e., factor of five to ten in practice), thus the text positions consume more memory than the labels will later on. If we only compute s required text positions during a superstep and then retrieve them, we need $\mathcal{O}(\frac{n}{sc})$ supersteps. Thus, we can decrease the memory overhead by increasing the number of supersteps that are required during the construction. This yields the following space-time trade-off (regarding the maximum amount of memory required during construction).

COROLLARY 3.1. *Given the* SA *and* LCP*-array, the cost of constructing the Patricia trie is* $\mathcal{O}(\frac{n}{c} + \frac{n}{c}G + \frac{n}{sc}L)$ *if we only allow* $s \lg n$ *bits additional space.*

3.2 Querying. The global trie GT is constructed for the set $\mathcal{S} = \{S_{\mathsf{SA}_1[1]}, S_{\mathsf{SA}_1[\frac{n}{c}]}, \ldots, S_{\mathsf{SA}_c[1]}, S_{\mathsf{SA}_c[\frac{n}{c}]}\}$ and a maximum pattern length of $|\mathsf{P}|_{\max}$. For any $i \in [1, c]$ the $2i$-th leaf corresponds to the lexicographically smallest suffix and the $2i+1$-th leaf corresponds to the greatest suffix represented by PE i. Querying GT is different from querying a trie as we do not want to find all occurrences of a pattern P, but want to find all PEs that represent P. We still follow the edges according to their label and the corresponding position in P until we have matched P at a node u or have a mismatch with the label of an edge (v, w).

In the first case and if v is an internal node, we need to identify the leftmost and rightmost leaves below v. Let k and ℓ be the leaf ranks of those leaves, resp. Then all PEs j with $j \in [\lfloor \frac{k}{2} \rfloor, \lfloor \frac{\ell}{2} \rfloor]$ contain positions where the pattern occurs, as the PEs cannot be distinguished by P. If (in the first case) v is a leaf with leaf rank k, then PE $\lfloor \frac{k}{2} \rfloor$ can be responsible for P. In this case we cannot be sure, as $\mathsf{pathlabel}(v)$ may be a prefix of P. Therefore, we send P to PE $\lfloor \frac{k}{2} \rfloor$ and use the local Patricia trie $\mathsf{PT}_{\lfloor \frac{k}{2} \rfloor}$ to determine whether P occurs.

In the second case (there was a mismatch), P can still occur. Let α and β be the mismatching characters of the label and the pattern, resp. If $\alpha >_{\text{lex}} \beta$ we look at the leftmost leaf below w. If the leaf rank k is even, P does not occur in any PE, as it is smaller than the lexicographically smallest suffix represented by PE $\lfloor \frac{k}{2} \rfloor$ and greater than the lexicographically greatest suffix represented by PE $\lfloor \frac{k}{2} \rfloor - 1$ because otherwise another edge would be followed in the beginning. If the rank is odd, P may occur in PEs $\lfloor \frac{k}{2} \rfloor$. In the other case ($\alpha <_{\text{lex}} \beta$), we need to get the rightmost leaf below w and check the leaf rank. There may be an occurrence if the leaf rank is even and there cannot be an occurrence if the leaf rank is odd (with the same type of argumentation given before). All PEs that are responsible for a pattern P form a consecutive interval that we denote by $\mathsf{GT}(\mathsf{P}) = [\ell, r]$.

LEMMA 3.4. *Given* GT *and a pattern* P*. Let* $\mathsf{GT}(\mathsf{P}) = [\ell, r]$*, if* $\ell \neq r$ *then* P *occurs at least once in PEs* ℓ *and* r *and* $\frac{n}{c}$ *times in PEs* j *for all* $j \in (\ell, r)$*.*

Now we take a look at how to answer *pattern matching* queries in the local Patricia tries. First, we look at the processing of a single query. Later, we show how the index can be used to answer a batch of queries.

Pattern Matching Queries. There are three types of pattern matching queries that the DPT supports:

Existential: Given a pattern P, we want to know whether the pattern P occurs in the text T.

Counting: Given a pattern P, we want to know how often the pattern P occurs in the text T.

Enumeration: Given a pattern P, we want to know all text positions in T where P occurs.

First, we look at an existential query P that arrives at PE i. We can answer the query in three supersteps (see also Figure 2):

1. At PE i, we identify all PEs that are responsible for P, i.e., all PEs j with $j \in \mathsf{GT}(\mathsf{P})$. If $\ell \neq r$ we know that P occurs in T (see Lemma 3.4), else we send P to PE ℓ.

2. Next, we perform a blind search in PT_ℓ. If the blind search fails, we know that P does not occur in T. Otherwise, the blind search returns a text position q. During the communication phase we retrieve $\mathsf{T}[q..q + |\mathsf{P}| - 1]$.

3. Using $\mathsf{T}[q..q+|\mathsf{P}|-1]$ we can verify the existence of P in T in the third superstep. When a query can be answered at a PE, we do not send it somewhere else, as the target depends on the application the index is used for.

The cost of an existential query is the following. During the first superstep we identify all PEs that can answer the query and send it to one PE costing $\mathcal{O}(t_{\text{trie}}(\mathsf{P}) + |\mathsf{P}|\, G + L)$. We let $t_{\text{trie}}(\mathsf{P})$ denote the time required to search for P in a trie. Depending on the implementation this requires $\mathcal{O}(|\mathsf{P}| \lg \sigma)$ time (binary search) or $\mathcal{O}(|\mathsf{P}| + \lg\lg\sigma)$ time [9]. In the second superstep we perform a blind search and retrieve a substring of length $|\mathsf{P}|$. This costs $\mathcal{O}(t_{\text{trie}}(\mathsf{P}) + |\mathsf{P}|\, G + L)$. During the last superstep we just compare two strings of length $|\mathsf{P}|$ in $\mathcal{O}(|\mathsf{P}|)$ time.

Counting all occurrences of a pattern can be seen as an extension of the existential query and can be answered similarly (requiring four supersteps). Let P be a counting query arriving at PE i.

1. First, we identify all PEs that are responsible for P, i.e., all PEs j with $j \in \mathsf{GT}(\mathsf{P})$. Let all PEs j with $j \in [\ell, r]$ be responsible for P. If $\ell \neq r$ we know that P occurs in all those PEs (see Lemma 3.4). During the communication phase we send two queries Q_ℓ and Q_r to PE ℓ and r, resp. The former asks for the lexicographically smallest occurrence of P in PT_ℓ and the latter asks for the lexicographically largest occurrence of P in PT_r.

2. In the next step, we perform one blind search in PT_ℓ and one blind search in PT_r. If $\ell \neq r$ we know that the blind searches will return two text positions q_ℓ and q_r that are the lexicographically smallest and largest occurrences of P in T. If one of the blind searches fails we know that the PE is not responsible for P and we can send that there are no occurrences at the PE. During the communication phase, we retrieve $\mathsf{T}[q_\ell..q_\ell + |\mathsf{P}| - 1]$ and $\mathsf{T}[q_r..q_r + |\mathsf{P}| - 1]$.

3. Using $\mathsf{T}[q_\ell..q_\ell + |\mathsf{P}| - 1]$ and $\mathsf{T}[q_r..q_r + |\mathsf{P}| - 1]$ we can verify the existence of P in T (only necessary if $\ell \neq r$) and also find the number of occurrences at PEs ℓ and r using the leaf ranks. We send the number to PE i.

4. We know the number of occurrences occ_ℓ and occ_r of P in PEs ℓ and r, resp. We also know that P has to occur $\frac{n}{c}$ times at each PE j for $j \in (\ell, r)$. Thus the total number of occurrences of P is $occ_\ell + occ_r + \max(0, r - \ell - 1)\, \frac{n}{c}$.

The first superstep costs $\mathcal{O}(t_{\text{trie}}(\mathsf{P}) + |\mathsf{P}|\, G + L)$ as we need to identify the PEs that are responsible for the pattern and send it to two PEs. In the second superstep we perform a blind search at two PEs and retrieve two substrings of length $|\mathsf{P}|$. This costs $\mathcal{O}(t_{\text{trie}}(\mathsf{P}) + |\mathsf{P}|\, G + L)$. The third superstep consist of comparing the retrieved substrings with the pattern and send the number of occurrences of the pattern to the PE where the pattern arrived initially, i.e., PE i. This costs $\mathcal{O}(|\mathsf{P}| + G + L)$. During the last superstep we need to compute the total number of occurrences at PE i which costs $\mathcal{O}(1 + L)$.

Last, we consider enumeration queries. Let P be a enumeration query arriving at PE i. During the first two supersteps answering an enumeration query does not differ from answering a counting query. The remaining steps are changed as follows.

3. Using $\mathsf{T}[q_\ell..q_\ell + |\mathsf{P}| - 1]$ and $\mathsf{T}[q_r..q_r + |\mathsf{P}| - 1]$ we can verify the existence of P in T (only necessary if $\ell \neq r$) and also find all positions where the pattern occurs. We send all these positions to PE i.

4. We have received all occurrences of P from the PEs ℓ and r. Next we need to retrieve all occurrences, i.e., the local SA from all PEs j for $j \in (\ell, r)$.

The first two supersteps are the same as for a counting query. Therefore, the costs of the first two supersteps are the same. The third superstep is very similar to the third superstep for answering a counting query. The only difference is that we need to send the text positions of all occurrences to the PE where the query arrived initially. This costs $\mathcal{O}(|\mathsf{P}| + occ \cdot G + L)$, where occ denotes the number of occurrences of P at PE ℓ and PE r. Last, we need to retrieve the text positions of all occurrences of P in PEs j with $j \in (\ell, r)$, which costs $\mathcal{O}\big((r - \ell)\frac{n}{c} G + L\big)$ if $\ell < r + 1$.

Answering any type (existential, counting or enumeration) of query has (asymptotically) the same cost for the first two supersteps, as we send at most twice as many queries to the second level (for counting and enumeration queries). To answer counting queries we send the leaf ranks during third superstep yielding a cost

of $\mathcal{O}(|\mathsf{P}| + G + L)$. In the last superstep we just need to add up the number of occurrences in $\mathcal{O}(1)$ time. When we consider enumeration queries, we need to report all text positions where P occurs. Let *occ* be the maximum number of occurrences of P in a PE j for $j \in [\ell, r]$, then the cost of the third superstep is $\mathcal{O}(|\mathsf{P}| + occ \cdot G + L)$. In the fourth superstep we need to retrieve all positions from the PEs j for all $j \in (\ell, r)$ costing $\mathcal{O}(1 + c \cdot occ \cdot G + L)$. All in all we get the following costs.

LEMMA 3.5. *Given a pattern* P *answering an existential query costs* $\mathcal{O}(t_{trie}(\mathsf{P}) + |\mathsf{P}|\, G + L)$*, answering a counting query costs* $\mathcal{O}(t_{trie}(\mathsf{P}) + |\mathsf{P}|\, G + L)$*, and answering an enumeration query costs* $\mathcal{O}(t_{trie}(\mathsf{P}) + (|\mathsf{P}| + occ)\, G + L)$*, where occ denotes the total number of occurrences of* P*.*

Batched Queries and Load Balancing. When we process a batch of q queries at once rather than a single query, the number of supersteps does not increase, i.e., we can amortize the startup latencies of the BSP model over a large number of queries. Moreover, if the local work and communication volume is well balanced over the PEs, the query throughput scales linearly with c. On the one hand, we can measure empirically how well balanced the computation is – see also § 4. On the other hand, we can see to what extent good balance can be enforced.

Balancing the queries itself can be achieved using any standard load balancing technique, i.e., assuming that $\mathcal{O}(q/c)$ queries arrive at each PE is unproblematic.

Balancing how many queries get directed at each local trie is more difficult, since certain patterns might be more popular than others. However, we can use "virtualization" – we split the corpus into $c' \gg c$ pieces and distribute them randomly to the actual PE.

Similarly, some documents might be more popular than others. However, by randomly permuting the documents in the corpus, we can at least ensure that it is unlikely that many popular documents are assigned to the same PE.

When all these balancing conditions are fulfilled, a batch Q of queries can be completed in time

$$\mathcal{O}\left(\frac{1}{c}\left(\sum_{\mathsf{P}\in Q} t_{\text{trie}}(\mathsf{P}) + (|\mathsf{P}| + occ(\mathsf{P}))G\right) + L\right),$$

where $occ(\mathsf{P})$ denotes the number of occurrences of P for an enumeration query (and 0 else).

Comparison to the Distributed Suffix Array. Using the (multiplexed) distributed suffix array (DSA) [1], a batch Q of q counting queries can be answered in time

$$\mathcal{O}\left(\frac{1}{c}\left(\sum_{\mathsf{P}\in Q} t_{\text{Bin}}(\mathsf{P}) + (|\mathsf{P}| \lg \frac{n}{q} + \lg c)G\right) + \lg \frac{cn}{q} L\right),$$

where $t_{\text{Bin}}(\mathsf{P})$ denotes the time to identify the occurrences of the pattern P in the SA, i.e., $|\mathsf{P}| \lg n$. Distributing the queries costs $\mathcal{O}\Big(\sum_{\mathsf{P}\in Q} |\mathsf{P}| + |\mathsf{P}|\, G + L\Big)$ and is dominated by the costs of answering the batch of queries.

Comparing the costs of the DSA with our DPT we get the following result: The maximum time used for computation by each PE is $\mathcal{O}\Big(\frac{1}{c}\Big(\sum_{\mathsf{P}\in Q} t_{\text{trie}}(\mathsf{P})\Big)\Big)$ using the DPT since we look in GT and at most two local PTs for each query. The computation time required by the DSA is $\mathcal{O}\Big(\frac{1}{c}\Big(\sum_{\mathsf{P}\in Q} t_{\text{Bin}}(\mathsf{P})\Big)\Big)$ and results from the binary searches (local and inter-PE). Hence, the time used for computation by each PE differs with respect to the time required for searching the corresponding suffix array interval for each pattern using a trie and using binary search. Usually, we can assume that $t_{\text{trie}}(\mathsf{P})$ is smaller than $t_{\text{Bin}}(\mathsf{P})$.

The cost of communication is $\mathcal{O}\Big(\frac{1}{c}\Big(\sum_{\mathsf{P}\in Q}(|\mathsf{P}|)\Big)\, G\Big)$ using the DPT, as we just send each pattern to at most two PEs and retrieve a substring of the length of the pattern. For the DSA the cost of communication is higher, i.e., $\mathcal{O}\Big(\frac{1}{c}\Big(\sum_{\mathsf{P}\in Q} |\mathsf{P}| \lg \frac{n}{q} + \lg c\Big)\, G\Big)$ because more substrings need to be retrieved during the binary search. This effect can be moderated by storing pruned suffixes for each position of SA – see § 4.3. Still, the DPT requires only a constant number of substrings to be retrieved for each query.

Last, the synchronization using DPT is constant, i.e., $\mathcal{O}(L)$, but using the DSA synchronization costs $\mathcal{O}\Big(\lg \frac{cn}{q} L\Big)$. Due to the constant number of messages being sent using the DPT, the synchronization cost is optimal and a logarithmic factor worse using the DSA.

Therefore, if we assume an optimal distribution of the queries and of the documents, the DPT is theoretically faster than the DSA. This difference in cost can also be seen in practice – see § 4. However, the multiplexed DSA is very strong against query bias, whereas the DPT can be affected by query bias resulting in a load imbalance.

4 Experiments

We implemented the distributed Patricia trie using C++ (`g++` 6.1 with flags `-O3 -march=native`). The communication is handled by the *Message Passing Interface* (MPI, Open MPI 1.10.3) with each MPI process representing a PE of our algorithm. For the representation of bit vectors and the operations rank(), select() and find_close(), we use the *succinct data structure library* (sdsl-lite, 2.0.1) [12]. In particular we used the RANK_SUPPORT_V5 for rank(), SELECT_SUPPORT_MCL for select() and BP_SUPPORT_SADA for find_close(). We computed the SA and LCP-array using the implementation of Flick et al. [11]. The source code of our implementation and data required to reproduce our results are available from `https://github.com/kurpicz/dpt`.

For our experiments we use the *common crawl corpus* as input.[1] It provides world wide web crawl data and contains raw content, text only and metadata of the crawled websites from about 1.23 billion web pages. In total the corpus has a size of 541 TB (as of 27.07.2016). We use the *WET files* that contain only the text without any tags or other meta information. (We removed all additional data added by the common crawl corpus.)

The data we use for queries comes from the following sources:

Text Retrieval Conference (TREC) Containing all published test queries of the *Million Query Track*, which contains 60k queries in total [26].

AOL Query Log (AOL) Contains around 20M web queries collected from roughly 650k users that have been collected over three months [27].

To the best of our knowledge, the distributed suffix array (DSA) presented by Arroyuelo et al. [1] is the only other (available) implementation of a distributed full-text index. Hence, we compare our results with the fastest variant of the DSA, the *multiplexed* distribution of the SA. The implementation of the DSA only supports counting queries. Therefore, in § 4.2 we only compare the time for counting queries. We adapted the (non-public) source code such that it works with texts of size greater than 4 GiB.

All experiments were conducted on the *InstitutsCluster II* (IC2) at KIT.[2] The cluster has compute nodes consisting of two Octa-Core Intel Xeon E5-2670 processors with 64 GB main memory and 2 TB external memory. The nodes are connected using InfiniBand QDR. Up to 32 nodes at a time were available for our measurements. Each node runs 16 MPI processes and in this section PE refers to MPI process.

4.1 Construction. As noted before, the construction of the SA and LCP-array are not part of this paper – we start the timer as soon as each PE holds its local share of the SA, LCP-array and T. Still, the construction of the SA and LCP-array are the bottleneck of our data structure. Figure 3 shows the construction time of the distributed Patricia trie using a *weak scaling* plot, i.e., for each curve the amount of input data per PE is fixed. In this case we used up to 100 GiB of text which translates to 200 MiB of text per PE and constructed the DPT for $|\mathsf{P}|_{\max} = 30$. We could not run bigger experiments as we were only able to compute the SA and LCP-array for texts up to that size using the available resources and the only distributed construction algorithm that can compute both arrays, i.e., [11].

The construction time for the SA and LCP-array is two to three times greater than the time required for the construction of the DPT. Hence, we can say that our construction time is reasonable and practical.

We compare two different variants of implementation: Using the collective communication operation MPI_Alltoallv and using a large batch of *remote direct memory access* operations (RDMA, operation MPI_Get). Preparing the requests and constructing the tree topology are the most time consuming tasks during the construction of the DPT. One-sided communication turns out to be somewhat faster since it does not need to send text positions first, but can directly access the text. The amount of messages sent and received by all PEs is (with a few small exceptions) homogeneously distributed among all PEs – see Figure 5 (left).

The space consumption of our index can be seen in Table 1, where we compare the different representations of the local Patricia tries. Since we need to keep track of the nodes on the stack, the memory usage is higher

[1] `http://commoncrawl.org/`

[2] `https://www.scc.kit.edu/dienste/ic2.php`

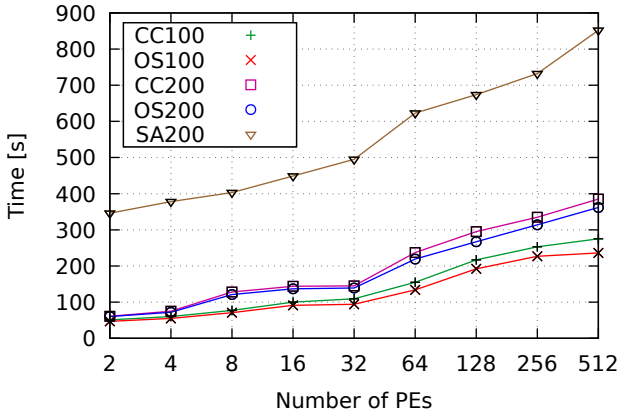


Figure 3: Construction times of the DPT for 100 MiB (CC100 and OS100) and 200 MiB (CC200 and OS200) of text per PE, utilizing collective communication (CC) and one-sided communication (OS). SA200 is the construction time of the SA and LCP-array for 200 MiB text per PE.

	Pointer	BP	DFUDS	LOUDS
peak	$46n$	$19n$	$18n$	$18n$
size	$42n$	$16n$	$16n$	$15n$

Table 1: Bits needed to store the local tries during construction (peak) and querying (size) using 40-bit text positions.

during the construction. In the following, we omit a detailed analysis of DFUDS and BP, as LOUDS is better with respect to size and speed.

The multiplexed distributed suffix array by Arroyuelo et al. [1] requires preprocessing of SA and T. During the preprocessing, the multiplexed SA for each PE is constructed and pruned suffixes for each text positions are stored accordingly (see § 4.3 for the effect of pruning in practice). We refer to this preprocessing as construction time. However, the process is not distributed; one PE prepares all required files. Therefore, we omit a comparison of the construction time with DPT as it does not scale.

A standard question for a parallel algorithm is about its speedup with respect to the best sequential algorithm. This is important for understanding how much overhead is involved in going to a distributed environment. Since the SA and LCP-array construction dominates index construction time and since there is no tuned sequential implementation of DPT construction itself, we make this comparison only for the SA and LCP-array construction. The fastest sequential algorithm we are aware of is *divsufsort*, which computes the SA (but not the LCP array).[3] Running the 64-bit version of divsufsort 2.0.1 for 50 GiB on a machine with 512 GB RAM and 4 Intel Xeon E5-4640 processors takes 42 247 seconds. Extrapolating this to 100 GiB gives 84 494 seconds – about 100 times more than the algorithm from [11]. Note that divsufsort is much slower when extrapolating from traditionally small inputs as we need a 64-bit version, due to NUMA effects, and because for really large inputs a logarithmic term in virtual address translation becomes noticeable [16]. There exists also a parallel variant of divsufsort, where one of its two sorting steps is parallelized. Hence, not the whole algorithm is executed in parallel. Therefore, on 50 GiB and using 32 cores, it is only about 25% faster than sequential divsufsort.

Another interesting comparison is with a state of the art general purpose external memory algorithm [3]. Here, the algorithm from [11] on 512 processors is about 280 times (using somewhat faster processors but on larger inputs than [3]).

[3] `https://github.com/y-256/libdivsufsort`

4.2 Query Time. In this section we focus on pattern matching queries. Once more, we perform a weak scaling experiment to compare the times required to answer batches of queries. This time, both the input size per PE (200 MiB) and the number q of queries arriving at each PE (20k, 40k and 80k) are fixed. Thus, we build an index on up to 100 GiB of text, and want to answer a batch of at most 41M queries. If the number of queries exceeds the available number of queries (60k and 20M for TREC and AOL queries, resp.), we replicate the set of queries accordingly. Also, we stop the timing as soon as the result for a query is known at any PE. For example, if we have a counting query that can be answered by a single PE, we do not send the result to the PE where the query arrived initially.

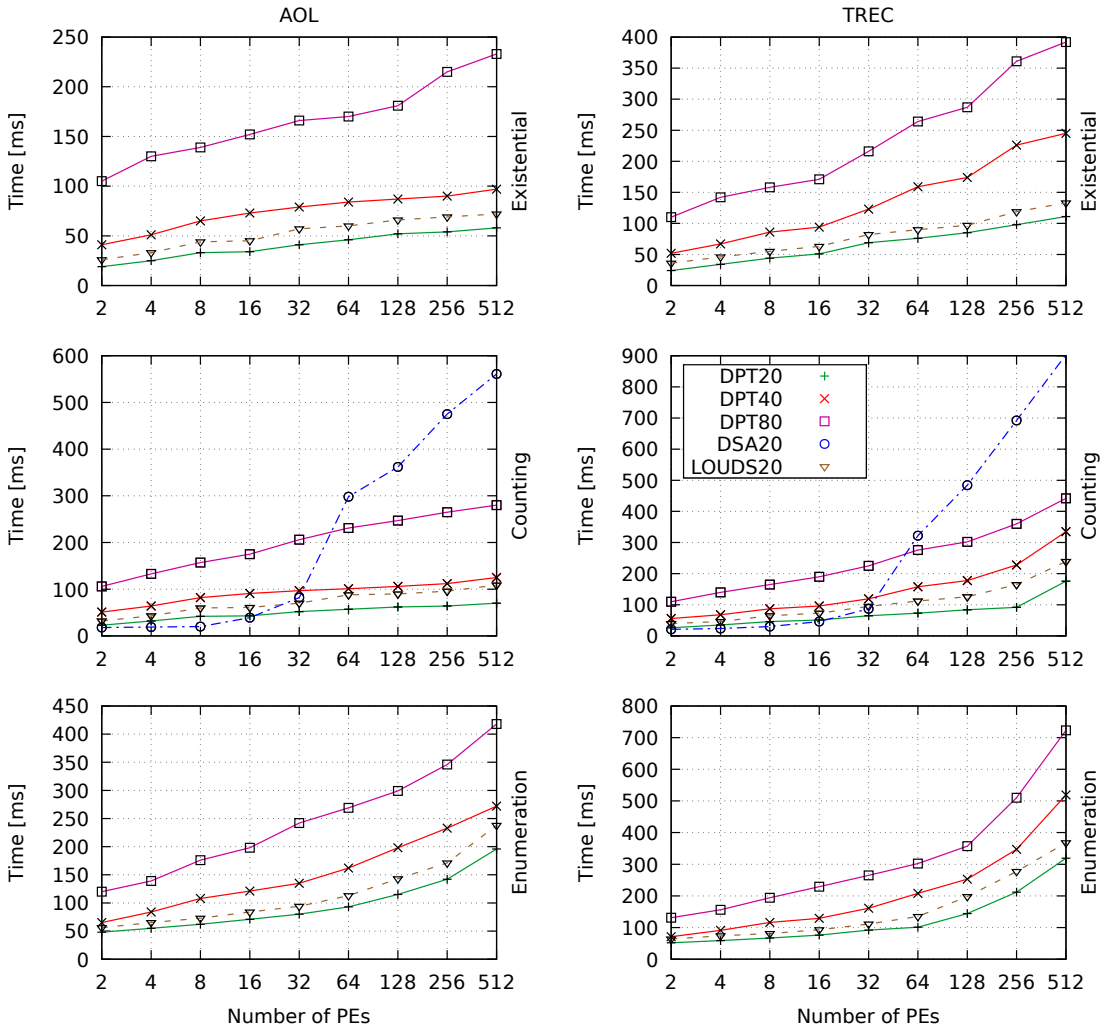


Figure 4: Query times of the DPT for 200 MiB of text per PE. Each PE receives 20k (DPT20), 40k (DPT40) and 80k (DPT80) queries. LOUDS20 is the query time for 20k queries using LOUDS as succinct trie representation and DSA20 is the query time for the distributed suffix array, also asking for 20k queries per PE.

Figure 4 shows the query times for existential, counting and enumeration queries. DPT20 represents the times required to answer a batch of 20k queries per PE, DPT40 and DPT80 denote the times for 40k and 80k queries, using the pointer-based representation of the tries. The labels DSA20 and LOUDS20 denote the times for 20k queries using the DSA and the succinct LOUDS representations of the DPT. We augment the DSA with pruned suffixes of size 5 – see § 4.3 for more details. All query types scale reasonably well with the number of PEs. Existential queries can be answered the fastest, as they only require a single blind search (including the retrieval of a substring). They can be answered up to twice as fast as the counting queries.

Counting queries require more work at each PE, as we need to compute the number of occurrences, which translates to two traversals from one node to a leaf at the local PT. Comparing our implementation with the DSA we see that we scale better as we can send our queries to the PE that can actually answer them. This allows us to answer queries up to 5.5 times faster. The implementation of DSA only supports counting queries, therefore, DSA only appears for counting queries in Figure 4. When utilizing more than 32 PEs, DSA becomes significantly slower than DPT. This is probably due to the increasing number of messages sent by the index that are necessary during the binary search, whenever the pattern must be compared with the text. For larger texts, there are fewer text positions that correspond to text that is locally available. This effect can be reduced by *pruning* (see § 4.3).

Last, we have enumeration queries, which are the hardest to answer as we have to compute all text positions. They can take up to 1.5 times as long as counting and 3 times as long as existential queries.

The TREC queries are generally harder to answer as there are fewer and we have duplicates as soon as we require more than 60k queries. Duplicates lead to a higher imbalance regarding the query distribution, i.e., query bias – see Figure 5 (right) for the query distribution among the PEs.

The succinct tree representations are reasonably fast compared with the pointer based implementation – see again Figure 4. LOUDS is the fastest of the three tested succinct tree representations, being only 10% to 20% slower than the pointer based implementation but using only about a third of the space. This means that succinct data structures allow us to use 2.5 times less memory to build the DPT for the same text as a pointer based implementation. Counting and enumeration queries are more expensive when it comes to succinct tree structures, as we need to traverse the trie multiple times if the pattern occurs.

4.3 Distributed Suffix Array – Pruning. In addition to the local multiplexed SA, the (multiplexed) DSA also holds *pruned suffixes*, i.e., for each position in the multiplexed SA, the first ℓ characters of the corresponding suffix are stored. Since the text corresponding to a suffix may not be available locally, this speeds up the query time (by increasing the size of the index). We tested different sizes ℓ for the pruned suffixes, using 200 MiB of text per PE and asking for 20k AOL queries per PE, i.e., the same configuration as in our experiments for the query times. The speedup is listed in Table 2.

PEs	ℓ	speedup	PEs	ℓ	speedup
2	0	1	8	0	1
	5	1.04		5	1.13
	10	1.20		10	1.51
	15	1.28		15	1.54
	20	1.33		20	1.71
4	0	1	16	0	1
	5	1.33		5	1.05
	10	1.42		10	1.67
	15	1.41		15	1.75
	20	1.45		20	1.92

Table 2: Speedup of the query time in the DSA with respect to the size ℓ of the pruned suffixes.

Since the average size of an AOL query is 18, pruned suffixes of size greater than 20 do not provide any more significant speedup. In our experiments we used pruned suffixes of size 5, as this adds 40 bytes to the DSA per text position, which corresponds to the size of our pointer based DPT representation. Choosing larger pruned suffixes can speedup the DSA such that it is faster than the DPT on 32 PEs. However, this also results in an index that is larger than our DPT.

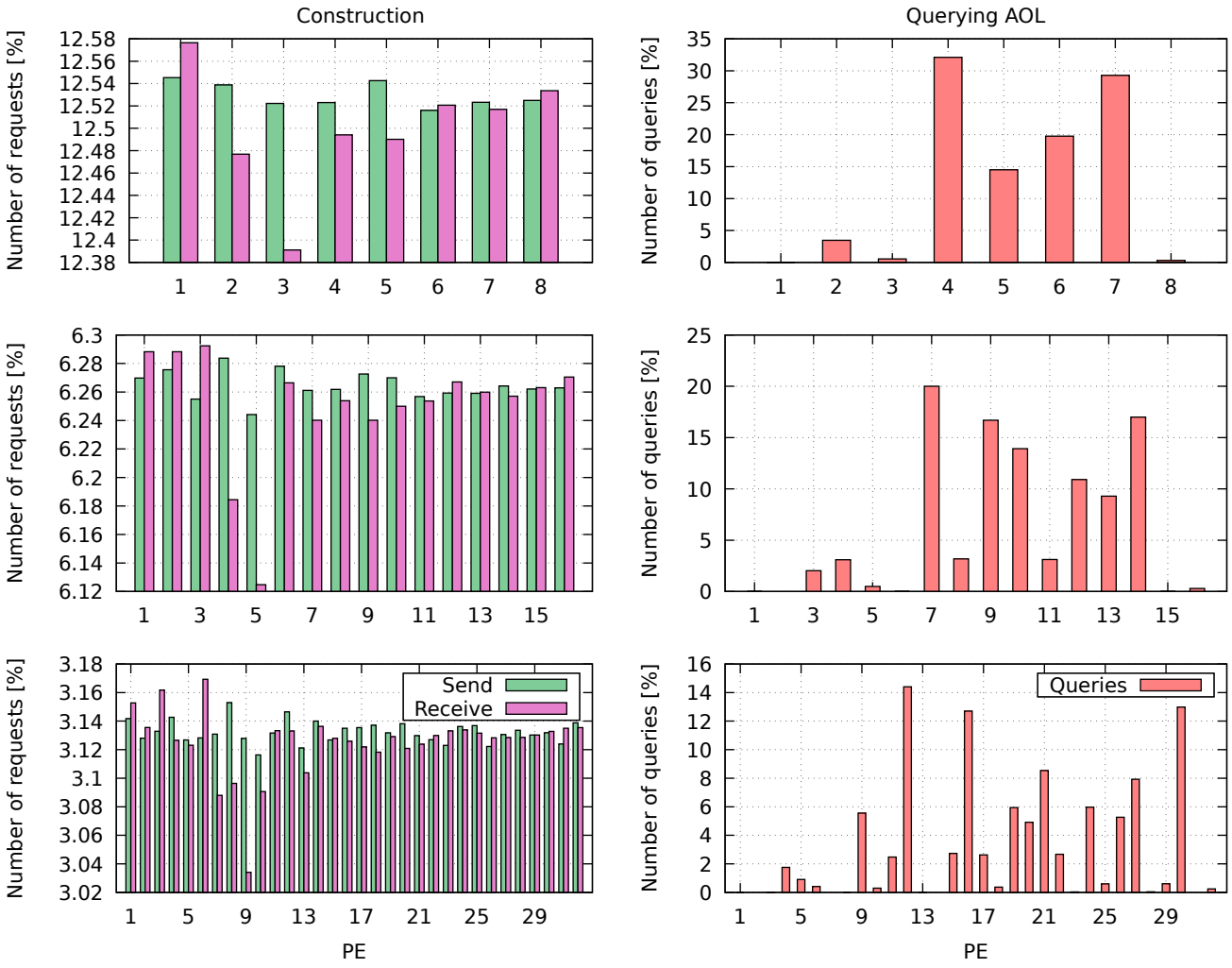


Figure 5: Work imbalance between different PEs. For each PE, we give the percentage of requests sent and received during construction of the DPT and percentage of queries received during querying by each PE. Here, the DPT is constructed for 200 MiB of text per PE and we ask for 20k AOL queries per PE.

5 Conclusion and Future Work

We presented a distributed full-text index that supports existential, counting and enumeration queries. All queries can be answered using a constant number of messages with length proportional to the queries. Our implementation scales well regarding the query processing and is faster than the DSA when run on more than 32 PEs. For 512 PEs our index can answer a batch of counting queries up to 5.5 times faster than the DSA. Also, we use succinct data structures, resulting in lower space requirements by a factor of 2.5 at only about 20% slowdown.

Still, there are optimizations that may lead to better performance in the future. One important issue is to further develop and implement the load balancing measures outlined in § 3.2. An orthogonal issue is to consider a stream of queries, i.e., instead of a batch of queries arriving at each PE at the same time, we assume that queries can arrive at any PE at any time. This is a real world problem, as for many applications, queries do not arrive in a batch. For this setting, we need asynchronous communication between the PEs. In this scenario, an interesting question is at which amount of queries batched query processing becomes more efficient than the streaming mode.

In addition, our index can be extended to also answer document retrieval queries [25], where the text is composed of a number of (short) documents, and one wishes to count or enumerate all *documents* containing a given query pattern (documents containing the pattern multiple times should only be counted/enumerated *once*). For document counting, one could use the preprocessing by Hui [14], while for optimal document listing, the technique of Hon et al. [13] could be adapted. This latter technique relies on a data structure for fast range minimum queries [10], which has to be transformed into a distributed environment.

Another interesting direction is the usage of compressed cache-oblivious tries as shown by Ferragina and Venturini [8]. Further improvements should also be possible using hybrid parallelism, i.e., to exploit that PEs on the same compute node can quickly interact using shared memory.

6 Acknowledgments

We would like to thank the authors of [1] for providing their source code, and in particular Veronica Gil-Costa for answering all questions regarding the implementation. We also want to thank Timo Bingmann for providing numbers on sequential suffix array construction. Our research was supported by the German Research Foundation (DFG), priority programme "Algorithms for Big Data" (SPP 1736).

References

[1] Diego Arroyuelo, Carolina Bonacic, Veronica Gil-Costa, Mauricio Marin, and Gonzalo Navarro. Distributed text search using suffix arrays. *Parallel Computing*, 40(9):471–495, 2014.

[2] David Benoit, Erik D. Demaine, J. Ian Munro, Rajeev Raman, Venkatesh Raman, and S Srinivasa Rao. Representing trees of higher degree. *Algorithmica*, 43(4):275–292, 2005.

[3] Timo Bingmann, Johannes Fischer, and Vitaly Osipov. Inducing suffix and lcp arrays in external memory. *J. Exp. Algorithmics*, 21:2.3:1–2.3:27, 2016.

[4] David Clark. *Compact Pat Trees*. PhD thesis, University of Waterloo, 1996.

[5] Raphaël Clifford. Distributed suffix trees. *J. Discrete Algorithms*, 3(2-4):176–197, 2005.

[6] Paolo Ferragina and Roberto Grossi. The string b-tree: a new data structure for string search in external memory and its applications. *J. ACM*, 46(2):236–280, 1999.

[7] Paolo Ferragina and Fabrizio Luccio. String search in coarse-grained parallel computers. *Algorithmica*, 24(3-4):177–194, 1999.

[8] Paolo Ferragina and Rossano Venturini. Compressed cache-oblivious string b-tree. In *Annual European Symposia on Algorithms (ESA)*, volume 8125 of *LNCS*, pages 469–480. Springer, 2013.

[9] Johannes Fischer and Pawel Gawrychowski. Alphabet-dependent string searching with wexponential search trees. In *Annual Symposium on Combinatorial Pattern Matching (CPM)*, volume 9133 of *LNCS*, pages 160–171, 2015.

[10] Johannes Fischer and Volker Heun. Space efficient preprocessing schemes for range minimum queries on static arrays. *SIAM J. Comput.*, 40(2):465–492, 2011.

[11] Patrick Flick and Srinivas Aluru. Parallel distributed memory construction of suffix and longest common prefix arrays. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 16:1–16:10. ACM, 2015.

[12] Simon Gog, Timo Beller, Alistair Moffat, and Matthias Petri. From theory to practice: Plug and play with succinct data structures. In *International Symposium on Experimental Algorithms (SEA)*, volume 8504 of *LNCS*, pages 326–337. Springer, 2014.

[13] Wing-Kai Hon, Rahul Shah, Sharma V. Thankachan, and Jeffrey Scott Vitter. Space-efficient frameworks for top-k string retrieval. *J. ACM*, 61(2):9:1–9:36, 2014.

[14] Lucas Chi Kwong Hui. Color set size problem with application to string matching. In *Annual Symposium on Combinatorial Pattern Matching (CPM)*, volume 644 of *LNCS*, pages 230–243. Springer, 1992.

[15] Guy Jacobson. Space-efficient static trees and graphs. In *Annual Symposium on Foundations of Computer Science (FOCS)*, pages 549–554. IEEE Computer Society, 1989.

[16] Tomasz Jurkiewicz and Kurt Mehlhorn. On a model of virtual address translation. *J. of Experimental Algorithmics*, 19:1.9:1–1.9:28, 2015.

[17] Juha Kärkkäinen, Peter Sanders, and Stefan Burkhardt. Linear work suffix array construction. *J. ACM*, 53(6):1–19, 2006.

[18] Fabian Kulla and Peter Sanders. Scalable parallel suffix array construction. *Parallel Computing*, 33:605–612, 2007. Special issue on Euro PVM/MPI 2006, distinguished paper.

[19] N. Jesper Larsson and Kunihiko Sadakane. Faster suffix sorting. *Theor. Comput. Sci.*, 387(3):258–272, 2007.

[20] Veli Mäkinen, Djamal Belazzougui, Fabio Cunial, and Alexandru I. Tomescu. *Genome-Scale Algorithm Design: Biological Sequence Analysis in the Era of High-Throughput Sequencing*. Cambridge University Press, 2015.

[21] Veli Mäkinen, Gonzalo Navarro, and Kunihiko Sadakane. Advantages of backward searching – efficient secondary memory and distributed implementation of compressed suffix arrays. In *International Symposium on Algorithms and Computation (ISAAC)*, volume 3341 of *LNCS*, pages 681–692. Springer, 2004.

[22] Udi Manber and Eugene W. Myers. Suffix arrays: A new method for on-line string searches. *SIAM J. Comput.*, 22(5):935–948, 1993.

[23] Donald R. Morrison. PATRICIA - practical algorithm to retrieve information coded in alphanumeric. *J. ACM*, 15(4):514–534, 1968.

[24] J. Ian Munro and Venkatesh Raman. Succinct representation of balanced parentheses and static trees. *SIAM J. Comput.*, 31(3):762–776, 2001.

[25] Gonzalo Navarro. Spaces, trees, and colors: The algorithmic landscape of document retrieval on sequences. *ACM Comput. Surv.*, 46(4):Article No. 52, 2014.

[26] National Institute of Standards and Technology. Million query track, 2010. `http://trec.nist.gov/data/million.query.html` accessed 04.07.2016.

[27] Greg Pass, Abdur Chowdhury, and Cayley Torgeson. A picture of search. In *International Conference on Scalable Information Systems (INFOSCALE)*. ACM, 2006.

[28] Luís M. S. Russo, Gonzalo Navarro, and Arlindo L. Oliveira. Parallel and distributed compressed indexes. In *Symp. on Combinatorial Pattern Matching (CPM)*, volume 6129 of *LNCS*, pages 348–360. Springer, 2010.

[29] Kunihiko Sadakane and Gonzalo Navarro. Fully-functional succinct trees. In *Annual ACM-SIAM Symposium on Discrete Algorithms, (SODA)*, pages 134–149. SIAM, 2010.

[30] David B. Skillicorn, Jonathan M. D. Hill, and William F. McColl. Questions and answers about BSP. *Scientific Programming*, 6(3):249–274, 1997.

[31] Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, 1990.