Theoretical Computer Science 700 (2017) 1–22

Contents lists available at ScienceDirect

Theoretical Computer Science

www.elsevier.com/locate/tcs

Parallel construction of succinct trees ☆,☆☆

José Fuentes-Sepúlveda [a,*], Leo Ferres [b], Meng He [c], Norbert Zeh [c]

[a] Department of Computer Science, Universidad de Chile, Santiago, Chile
[b] Faculty of Engineering, Universidad del Desarrollo & Telefónica I+D, Santiago, Chile
[c] Faculty of Computer Science, Dalhousie University, Halifax, Canada

ARTICLE INFO

Article history:
Received 2 September 2016
Received in revised form 9 June 2017
Accepted 29 July 2017
Available online 3 August 2017
Communicated by C. Kaklamanis

Keywords:
Succinct data structure
Succinct tree construction
Multicore
Parallel algorithm

ABSTRACT

Succinct representations of trees are an elegant solution to make large trees fit in main memory while still supporting navigational operations in constant time. However, their construction time remains a bottleneck. We introduce two parallel algorithms that improve the state of the art in succinct tree construction. Our results are presented in terms of *work*, the time needed to execute a parallel computation using one thread, and *span*, the minimum amount of time needed to execute a parallel computation, for any amount of threads. Given a tree on n nodes stored as a sequence of balanced parentheses, our first algorithm builds a succinct tree representation with $O(n)$ work, $O(\lg n)$ span and supports a rich set of operations in $O(\lg n)$ time. Our second algorithm improves the query support. It constructs a succinct representation that supports queries in $O(c)$ time, taking $O(n + \frac{n}{\lg^c n}\lg(\frac{n}{\lg^c n}) + c^c)$ work and $O(c + \lg(\frac{nc^c}{\lg^c n}))$ span, for any positive constant c. Both algorithms use $O(n \lg n)$ bits of working space. In experiments using up to 64 cores on inputs of different sizes, our first algorithm achieved good parallel speed-up. We also present an algorithm that takes $O(n)$ work and $O(\lg n)$ span to construct the balanced parenthesis sequence of the input tree required by our succinct tree construction algorithm.

1. Introduction

Trees are ubiquitous in Computer Science. They have applications in every aspect of computing from XML/HTML processing to abstract syntax trees (AST) in compilers, phylogenetic trees in computational genomics or shortest path trees in path planning. The ever increasing amounts of structured, hierarchical data processed in many applications have turned the processing of the corresponding large tree structures into a bottleneck, particularly when they do not fit in memory. Succinct tree representations store trees using as few bits as possible and thereby significantly increase the size of trees that fit in memory while still supporting important primitive operations in constant time. There exist such representations that use only $2n + o(n)$ bits to store the topology of a tree with n nodes [2–8], which is close to the information-theoretic lower bound and much less than the space used by traditional pointer-based representations.

☆ A previous version of this paper appeared in the 14th International Symposium on Experimental Algorithms (SEA 2015) [1].

☆☆ This work was supported by the Emerging Leaders in the Americas scholarship programme, NSERC, and the Canada Research Chairs programme. The first author received funding from FONDECYT 3170534 and Basal Funds FB0001, CONICYT. The second author would like to thank Telefónica I+D Chile and the Chilean government initiative CORFO 13CEE2-21592 (2013-21592-1-INNOVA_PRODUCCION2013-21592-1) for financial support.

* Corresponding author.
E-mail addresses: jfuentess@dcc.uchile.cl (J. Fuentes-Sepúlveda), lferres@udd.cl (L. Ferres), mhe@cs.dal.ca (M. He), nzeh@cs.dal.ca (N. Zeh).

http://dx.doi.org/10.1016/j.tcs.2017.07.025
0304-3975/

Alas, the construction of succinct trees is quite slow compared to the construction of pointer-based representations. Multicore parallelism offers one possible tool to speed up the construction of succinct trees, but little work has been done in this direction so far. The only results we are aware of focus on the construction of wavelet trees, which are used in representations of text indexes. In [9], two practical multicore algorithms for wavelet tree construction were introduced. Both algorithms perform $O(n \lg \sigma)$[1] work and have $O(\lg n)$ span, where n is the input size, σ is the alphabet size, work is the time needed to execute a parallel computation using a single thread, and span is the minimum time needed to execute a parallel computation for any number of threads. In [10–12], Shun introduced new algorithms to construct wavelet trees in parallel. Among these algorithms, the best algorithm in practice performs $O(n \lg \sigma)$ work and has $O(\lg n \lg \sigma)$ span. Shun also explained how to parallelize the construction of rank/select structures so that it requires $O(n)$ work and $O(\lg n)$ span for rank structures, and $O(n)$ work and $O(\lg n)$ span for select structures.

In this paper, we provide a parallel algorithm that constructs the `RMMT` tree representation of [2] in $O(n)$ work and $O(\lg n)$ span. This structure is a simplified version of the succinct tree representation in [2], and it uses $2n + o(n)$ bits to store an ordinal tree on n nodes and supports a rich set of basic operations on these trees in $O(\lg n)$ time. While this query time is theoretically suboptimal, the `RMMT` structure is simple enough to be practical and has been verified experimentally to be very small and support fast queries in practice [13]. Combined with the fast parallel construction algorithm presented in this paper, it provides an excellent tool for manipulating very large trees in many applications.

We implemented and tested our algorithm on a number of real-world input trees having billions of nodes. Our experiments show that our algorithm run on a single core is competitive with state-of-the-art sequential constructions of the `RMMT` structure and achieves good speed-up on up to 64 cores and likely beyond.

We then designed a parallel algorithm to construct the more complex, optimal succinct tree representation that supports operations in $O(c)$ time using $2n + O(n/\lg^c n)$ bits, for any constant $c > 0$. This algorithm has $O(n + \frac{n}{\lg^c n} \lg(\frac{n}{\lg^c n}) + c^c)$ work and $O(c + \lg(\frac{nc^c}{\lg^c n}))$ span. In the design of this new algorithm, we provide parallel algorithms to construct the 2d-Min-Heap data structure and the ladder decomposition of trees. We think that those partial results may be of independent interest. For example, the ladder decomposition is used to support level ancestor in $O(1)$ time [14].

The remainder of this paper is organized as follows: Section 2 gives a brief overview of the `RMMT` structure, to clearly define its structure and illustrate how it can be used to support basic operations on trees efficiently. It also briefly discusses other previous work on succinct tree representations and reviews the dynamic multithreading model, which we use to analyze the theoretical running time of our algorithm. Section 3 describes our parallel algorithms for constructing succinct representations of trees and for computing the balanced parentheses representation of trees. Section 4 discusses our experimental setup and results. Section 5 offers concluding remarks and discusses future work.

2. Preliminaries

2.1. Succinct trees

Jacobson [3] was the first to propose the design of succinct data structures. He showed how to represent an ordinal tree on n nodes using $2n + o(n)$ bits so that computing the first child, next sibling or parent of any node takes $O(\lg n)$ time in the bit probe model. Clark and Munro [4] showed how to support the same operations in constant time in the word RAM model with word size $\Theta(\lg n)$. Since then, much work has been done on succinct tree representations, to support more operations, to achieve compression, to provide support for updates, and so on [15–19,5,6,2]. See [7] for a thorough survey.

Navarro and Sadakane [2] proposed a succinct tree representation, referred to as NS-representation throughout this paper, which was the first to achieve a redundancy of $O(n/\lg^c n)$ bits for any positive constant c. The *redundancy* of a data structure is the additional space it uses above the information-theoretic lower bound. While all previous tree representations achieved a redundancy of $o(n)$ bits, their redundancy was $\Omega(n \lg \lg n / \lg n)$ bits, that is, just slightly sub-linear. The NS-representation also supports a large number of navigational operations in constant time (see Table 1); only the work in [5,6] supports two additional operations: `level_leftmost`, which finds the leftmost node at a given level, and `level_successor(x)`, which finds the node immediately to the right of node x at the same level. An experimental study of succinct trees [13] showed that a simplified version of the NS-representation uses less space than other existing representations in most cases and performs most operations faster. In this paper, we provide a parallel algorithm for constructing this representation.

2.1.1. Simplified NS-representation

The NS-representation is based on the balanced parenthesis sequence P of the input tree T, which is obtained by performing a preorder traversal of T and writing down an opening parenthesis when visiting a node for the first time and a closing parenthesis after visiting all its descendants. Thus, the length of P is $2n$. See Fig. 1 as an example.

The NS-representation is not the first structure to use balanced parentheses to represent trees. Munro and Raman [15] used succinct representations of balanced parentheses to represent ordinal trees and reduced a set of navigational operations on trees to operations on their balanced parenthesis sequences. Their solution supports only a subset of the operations

[1] We use $\lg x$ to mean $\log_2 x$ throughout this paper.

Table 1
Operations supported by the NS-representation [2], including operations over the corresponding balanced parenthesis sequence.

No.	Operation	Description
1	`child`(x, i)	Find the ith child of node x
2	`child_rank`(x)	Report the number of left siblings of node x
3	`degree`(x)	Report the degree of node x
4	`depth`(x)	Report the depth of node x
5	`level_anc`(x, i)	Find the ancestor of node x that is i levels above node x
6	`subtree_size`(x)	Report the number of nodes in the subtree rooted at node x
7	`height`(x)	Report the height of the subtree rooted at x
8	`deepest_node`(x)	Find the deepest node in the subtree rooted at node x
9	`LCA`(x, y)	Find the lowest common ancestor of nodes x and y
10	`lmost_leaf`(x) /`rmost_leaf`(x)	Find the leftmost/rightmost leaf of the subtree rooted at node x
11	`leaf_rank`(x)	Report the number of leaves before node x in preorder
12	`leaf_select`(i)	Find the ith leaf from left to right
13	`pre_rank`(x)/`post_select`(x)	Report the number of nodes preceding node x in preorder/postorder
14	`pre_select`/`post_select`(i)	Find the ith node in preorder/postorder
15	`level_lmost`(i)/`level_rmost`(i)	Find the leftmost/rightmost node among all nodes at depth i
16	`level_succ`(x)/`level_pred`(x)	Find the node immediately to the left/right of node x among all nodes at depth i
17	`access`(i)	Report $P[i]$
18	`find_open`(i)/`find_close`(i)	Find The matching parenthesis of $P[i]$
19	`enclose`(i)	Find the closest enclosing matching parenthesis pair for $P[i]$
20	`rank_open`(i)/`rank_close`(i)	Report the number of opening/closing parentheses in $P[1..i]$
21	`select_open`(i)/`select_close`(i)	Find the ith opening/closing parenthesis

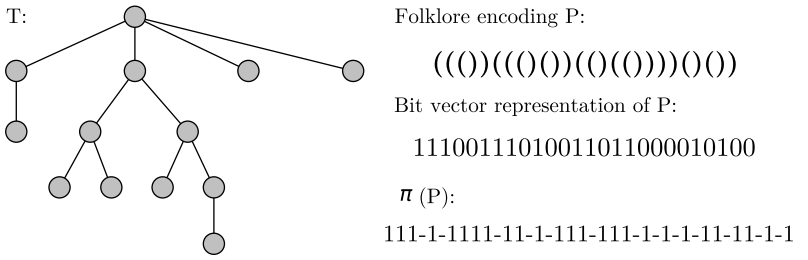


Fig. 1. Balanced parentheses representation P of a tree T. This representation, also known as folklore encoding, can be stored using a bit vector, writing a 1 for each opening parenthesis and a 0 for each closing parenthesis.

supported by the NS-representation. Additional operations can be supported using auxiliary data structures [19–22], but supporting all operations in Table 1 requires many auxiliary structures, which increases the size of the final data structure and makes it complex in both theory and practice. The main novelty of the NS-representation lies in its reduction of a large set of operations on trees and balanced parenthesis sequences to a small set of *primitive operations*. Representing P as a bit vector storing a 1 for each opening parenthesis and a 0 for each closing parenthesis (see Fig. 1), the following primitive operations can be defined, where g is an arbitrary function on $\{0, 1\}$:

$$\begin{aligned}
\texttt{sum}(P, g, i, j) &= \textstyle\sum_{k=i}^{j} g(P[k]) \\
\texttt{fwd_search}(P, g, i, d) &= \min\{j \mid j \geq i, \texttt{sum}(P, g, i, j) = d\} \\
\texttt{bwd_search}(P, g, i, d) &= \max\{j \mid j \leq i, \texttt{sum}(P, g, j, i) = d\} \\
\texttt{rmq}(P, g, i, j) &= \min\{\texttt{sum}(P, g, i, k) \mid i \leq k \leq j\} \\
\texttt{RMQ}(P, g, i, j) &= \max\{\texttt{sum}(P, g, i, k) \mid i \leq k \leq j\} \\
\texttt{rmqi}(P, g, i, j) &= \operatorname*{argmin}_{k \in [i,j]}\{\texttt{sum}(P, g, i, k)\} \\
\texttt{RMQi}(P, g, i, j) &= \operatorname*{argmax}_{k \in [i,j]}\{\texttt{sum}(P, g, i, k)\}
\end{aligned}$$

Most operations supported by the NS-representation reduce to these primitives by choosing g to be one of the following three functions:

$$\begin{array}{lll}
\pi : 1 \mapsto 1 & \phi : 1 \mapsto 1 & \psi : 1 \mapsto 0 \\
\quad 0 \mapsto -1 & \quad 0 \mapsto 0 & \quad 0 \mapsto 1
\end{array}$$

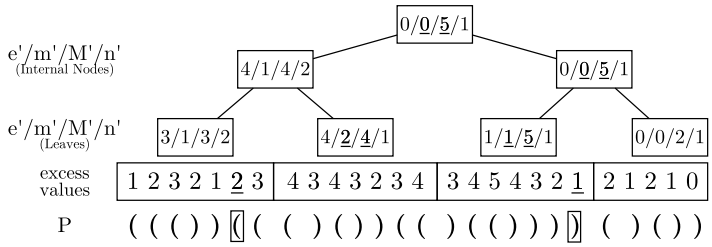


Fig. 2. Range min–max tree of the balanced parentheses sequence of the Fig. 1, with $s = 7$. In the figure, the m' and M' values involved in the operation $\texttt{fwd\_search}(P, \pi, 5, 0) = 20$ are underlined.

For example, assuming the ith parenthesis in P is an opening parenthesis, the matching closing parenthesis can be found using $\texttt{fwd\_search}(P, \pi, i, 0)$. Thus, it (almost)[2] suffices to support the primitive operations above for $g \in \{\pi, \phi, \psi\}$. To do so, Navarro and Sadakane designed a simple data structure called *Range Min–Max Tree* (RMMT), which supports the primitive operations above in logarithmic time when used to represent the entire sequence P. To achieve constant-time operations, P is partitioned into chunks. Each chunk is represented using an RMMT, which supports primitive operations inside the chunk in constant time if the chunk is small enough. Additional data structures are used to support operations on the entire sequence P in constant time.

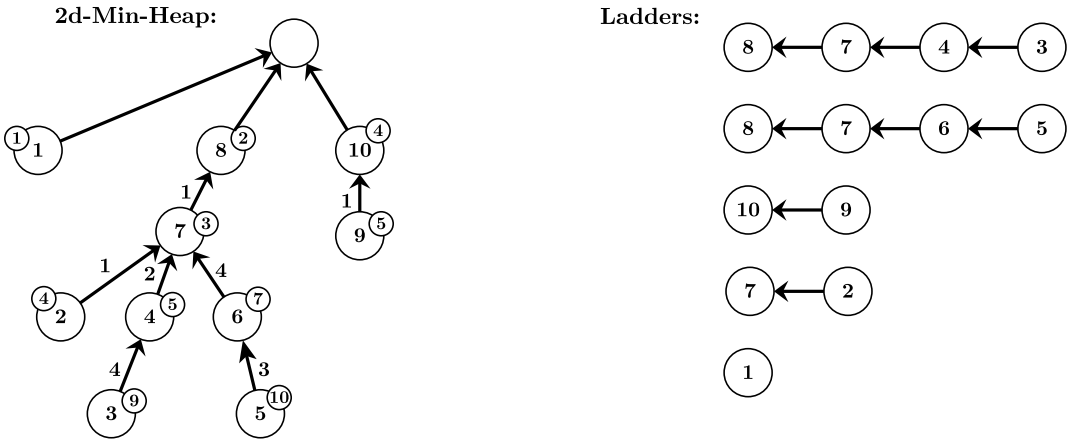
Next we briefly review the RMMT structure and how it supports the primitive operations for $g = \pi$ (see Fig. 1 for an example of function π). Navarro and Sadakane [2] discussed how to make it support these operations also for ϕ and ψ while increasing its size by only $O(n/\lg^c n)$. To define the variant of the RMMT we implemented, we partition P into chunks of size $s = w \lg n$, where w is the machine word size. For simplicity, we assume that the length of P is a multiple of s. The RMMT is a complete binary tree over the sequence of chunks (see Fig. 2). (If the number of chunks is not a power of 2, we pad the sequence with chunks of zeroes to reach the closest power of 2. These chunks are not stored explicitly.) Each node u of the RMMT represents a subsequence P_u of P that is the concatenation of the chunks corresponding to the descendant leaves of u. Since the RMMT is a complete tree, we need not store its structure explicitly. Instead, we index its nodes as in a binary heap and refer to each node by its index. The representation of the RMMT consists of four arrays e', m', M', and n', each of length equal to the number of nodes in the RMMT. The uth entry of each of these arrays stores some crucial information about P_u: Let the *excess* at position i of P be defined as $\texttt{sum}(P, \pi, 0, i) = \sum_{k=0}^{i} \pi(P[k])$. $e'[u]$ stores the excess at the last position in P_u. $m'[u]$ and $M'[u]$ store the minimum and maximum excess, respectively, at any position in P_u. $n'[u]$ stores the number of positions in P_u that have the minimum excess value $m'[u]$.

Combined with a standard technique called *table lookup*, an RMMT supports the primitive operations for π in $O(\lg n)$ time. Consider $\texttt{fwd\_search}(P, \pi, i, d)$ for example. We first check the chunk $j = \lfloor i/s \rfloor$ containing $P[i]$ to see if the answer is inside this chunk. This takes $O(\lg n)$ time by dividing the chunk into portions of length $w/2$ and testing for each portion in turn whether it contains the answer. Using a lookup table whose content does not depend on P, the test for each portion of length $w/2$ takes constant time: For each possible bit vector of length $w/2$ and each of the $w/2$ positions in it, the table stores the answer of $\texttt{fwd\_search}(P, \pi, i, d)$ if it can be found inside this bit vector, or -1 otherwise. As there are $2^{w/2}$ bit vectors of length $w/2$, this table uses $2^{w/2}\texttt{poly}(w)$ bits. If we find the answer inside the chunk j containing $P[i]$, we report it. Otherwise, we compute the global target $d' = e'[j] - \texttt{sum}(P, \pi, i, (j+1)s - 1) + d$. Let u be the leaf corresponding to this chunk. If u has a right sibling, we inspect the sibling's m' and M' values to determine whether it contains d'. If so, let v be this right sibling. Otherwise, we move up the tree from u until we find a right sibling v of an ancestor of u whose corresponding subsequence P_v contains the query answer. Then we use a similar procedure to descend down the tree starting from v to look for the leaf descendant of v containing the answer and spend another $O(\lg n)$ time to determine the position of d' inside its chunk. Since we spend $O(\lg n)$ time for each of the two leaves we inspect and the tests for any other node in the tree take constant time, the cost is $O(\lg n)$.

Fig. 2 shows the m' and M' values involved in the answer of $\texttt{fwd\_search}(P, \pi, 5, 0)$ (the matching closing parenthesis of the opening parenthesis at position 5). In this particular example, the objective is to find the closest position after $i = 5$ with excess value $d = 0$. Using lookup tables, we check if the answer is in the range $[5, 6]$ of the chunk $\lfloor 5/7 \rfloor = 0$. Since the answer is not there, we analyze the right sibling of the chunk 0 and compute the new global target $d' = 3 - 2 + 0 = 1$. The m' and M' values of the right sibling are 2 and 4, so d' is not there. We now move to the parent of the parent of the chunk 0. Let's call v to such node. The m' and M' values of v are 0 and 5, and therefore, the answer exists and it is in the right child of v. Then, we check the m' and M' values of the child of v. Those values are 1 and 5, and therefore, we need to move to the left child of the right child of v. Since the current node is a leaf, we use lookup tables to find the first value 1 is that chunk. In this case, such a 1 value is at position 20.

Supporting operations on the leaves, such as finding the ith leaf from the left, reduces to rank and select operations over a bit vector $P_1[0..2n-1]$ where $P_1[i] = 1$ iff $P[i] = 1$ and $P[i+1] = 0$. The $\texttt{rank}_c(P_1, i)$ operation counts the times

[2] A few navigational operations cannot be expressed using these primitives. The NS-representation includes additional structures to support these operations.



(a) *2d-Min-Heap* of the sequence $(1, 4, 9, 5, 10, 7, 3, 2, 5, 4)$.

(b) The *ladders* generated by the 2d-Min-Heap of Fig. 3a.

Fig. 3. Example of the sequence $(1, 4, 9, 5, 10, 7, 3, 2, 5, 4)$ and its ladders decomposition. In the 2d-Min-Heap, the indices of the sequence are inside large nodes, values are inside small circles and the weights of the edges are next to the edges.

symbol c appears up to position i in P_1. In turn, operation $\texttt{select}_c(P_1, j)$ returns the position in P_1 of the j-th appearance of symbol c, for $c \in \{0, 1\}$. `rank` and `select` operations over P_1 in turn reduce to `sum` and `fwd_search` operations over P_1 and can thus be supported by an `RMMT` for P_1. P_1 does not need to be stored explicitly because any consecutive $O(w)$ bits of P_1 can be computed from the corresponding bits of P using table lookup.

2.1.2. Constant time queries

To support constant time queries on arbitrary-sized trees, the balanced parentheses representation P needs to be partitioned into blocks. We represent each block using a `RMMT` and then construct additional data structures considering the minimum, maximum and excess values of the `RMMT` of each block. The size of each block is w^c, so we have $\tau = \lceil 2n/w^c \rceil$ of such blocks. To support constant time queries inside each block, we construct a `RMMT`, similar as before, but with chunk size $s = w/2$ and arity $k = \Theta(w/c \lg w)$, instead of arity 2. Let $m_1, \ldots, m_\tau$, $M_1, \ldots, M_\tau$, $n_1, \ldots, n_\tau$ and $e_1, \ldots, e_\tau$ be the minima, maxima, number of minima and excess stored at the root of the τ `RMMT`s. Depending on the operations, the additional data structures differ.

To solve $\texttt{fwd\_search}(P, \pi, i, d)$, we first try to solve it inside block $j = \lfloor i/w^c \rfloor$. The answer is returned if it is found in that block. If it is not, we must find the first excess $d' = d + e_{j-1} + \texttt{sum}(P, \pi, 0, i - 1 - w^c \cdot (j - 1))$ in the `RMMT`s of the following blocks. Applying Lemma 4.4 of [2], we must either find the first block $r > j$ such that $m_r \leq d'$, or such that $M_r \geq d'$. Once we find such a block, we complete the query inside of it with a local $\texttt{fwd\_search}(P, \pi, 0, d' - e_{r-1})$.

To find the corresponding block r in constant time, the authors propose additional data structures to represent the left-to-right minima and maxima values. For the case of left-to-right minima, it is necessary to build a tree called *2d-Min-Heap* (the left-to-right maxima is similar):

Definition 1. [2] Let $m_1, \ldots, m_\tau$ be a sequence of integers. We define for each $1 \leq j \leq \tau$ the left-to-right minima starting at j as $lrm(j) = (j_0, j_1, \ldots)$ where $j_0 = j$, $j_r < j_{r+1}$, $m_{j_{r+1}} < m_{j_r}$, and $m_{j_r+1}, \ldots, m_{j_{r+1}-1} \geq m_{j_r}$.

Once two *lrm* sequences coincide, they do so until the end. Thus, a 2d-Min-Heap is defined as a trie of τ nodes, composed of the reversed *lrm* sequences. Since the resulting trie can be composed of disconnected paths, a dummy root is used to generate the tree. If we assign weight to the edges, where the weight of an upward edge (j_i, j_{i+1}) is defined as $m_{j_i} - m_{j_{i+1}}$, we can reduce the problem of finding the first block $r > j$ such that $m_r \leq d'$ to a weighted level ancestor query over the 2d-Min-Heap. More precisely, we need to find the first ancestor j_r of node j such that the sum of the weights between j and j_r is greater than or equal to $d'' = m_j - d'$. Fig. 3a shows an example of the 2d-Min-Heap for the sequence $(1, 4, 9, 5, 10, 7, 3, 2, 5, 4)$.

To answer the weighted level ancestor query, we need to decompose the 2d-Min-Heap. The 2d-Min-Heap is decomposed into paths by recursively extracting the longest path. Then, for each path of length l, we store an extension of it by adding at most l nodes towards the root. These extended paths are called *ladders*. Fig. 3b shows an example of ladders. This decomposition ensures that a node with height h will have its first h ancestors in its ladder. For each ladder, a sparse bitmap is stored, where the i-th 1 of the bitmap represents the i-th node upward in the ladder, and the distance between two 1's is equal to the weight of the edge between them. All the bitmaps are concatenated into one of size $O(n)$, which is represented by the sparse bitmap of Pătraşcu [23]. Additionally, for each node v of the 2d-Min-Heap, the at most $\lg \tau$ ancestors at depths $depth(v) - 2^i$, $i \geq 0$ are stored in an array. Similarly, for each node v, the $\lg \tau$ accumulated weights toward the ancestors at distance 2^i are stored using *fusion trees* [24]. Fusion trees are used to store z keys of l bits each

one in $O(zl)$ bits, supporting predecessor queries in $O(\lg_l z)$ time, by using a $l^{1/6}$-ary tree. The $1/6$ factor can be reduced to achieve $O(1/\epsilon)$ predecessor query, where $0 < \epsilon \leq 1/2$ [2].

Observe that there is no guarantee that the weighted level ancestor j_r of the node j is in the ladder of j. Therefore, to answer the weighted level ancestor query we need first to compute the ancestor j' of node j with accumulated weight $2^{\lfloor \lg d'' \rfloor}$. The ancestor j' can be founded in constant time by a predecessor query of fusion tree of the node j and the array with the $\lg \tau$ ancestors of the node j. The answer is in the ladder of j'. If j' is at distance 2^i, then the answer is at distance less than 2^{i+1}. Applying rank/select queries over the bitmap of the ladder of node j', we find the node j_r.

To solve $\texttt{rmqi}(P, g, i, j)$ and $\texttt{RMQi}(P, g, i, j)$ operations on the τ blocks, we just need to build a data structure that supports range minimum and maximum queries in constant time, such as [25,26]. Both [25,26] use $2n + o(n)$ bits of extra space.

To solve $\texttt{degree}(i)$ operations, we need to consider *pioneers*. Let *pioneers* be the tightest matching pair of parentheses (i, j), with $j = \texttt{find\_close}(i)$, such that i and j belong to different blocks. We define a *marked* block to be a block that has the opening parenthesis of a pioneer (i, j) such that i and j do not belong to consecutive blocks. Let a be a marked block with pioneer (i, j) and let b be a block, we say that the block a *contains* the block b if the block b is between the blocks where i and j belong. We also say that b is a's child. There are $O(\tau)$ of such marked blocks. The $\texttt{degree}(i)$ operation, which computes the number of children of a node i, can be solved as follows: If the operation can be solved in at most two consecutive blocks, then the answer can be computed in constant time consulting the at most two corresponding RMMTs. Otherwise, it corresponds to the degree of a node that defines a marked block. Since there are $O(\tau)$ of such blocks, we can spend $O(\tau \lg n)$ bits to store explicitly the degree of all the nodes that define marked blocks and answer the operation in constant time.

The marked blocks are also used to support $\texttt{child}(i, q)$ and $\texttt{child\_rank}(i)$ operations. Both for $\texttt{child}(i, q)$ and $\texttt{child\_rank}(i)$, if the block of i is not a marked block, then both can be solved in at most two in-block queries. For marked blocks, we store a bitmap to represent the information about the children of each of the nodes that define them. For each marked block j, we store, in left-to-right order, information of marked blocks and blocks fully contained in j. For each block j' contained in block j, we store the number of children of the node associated to j that starts within j' (the number of minima of block j') and for each marked block contained in j, we store a 1, which represents a block containing one child of j. All numbers are stored in a bitmap as gaps of 0's between consecutive 1's. For the $\texttt{child}(i, q)$ query, we first check if $\texttt{child}(i, q)$ lies in the block of i or in $\texttt{find\_close}(i)$. If it does, we solve it with an in-block query. If not, we compute $p = \texttt{rank}_1(C_i, \texttt{select}_0(C_i, q))$, where C_i is the bitmap associated to the block of i. The value p represents the position of the block or marked block contained in i, where the q-th child of i lies. If it is a marked block, then it is the answer. If it is a block j, then the answer corresponds to the q'-th minimum within that block, where $q' = q - \texttt{rank}_0(C_i, \texttt{select}_1(C_i, p))$. $\texttt{child\_rank}(i)$ can be solved similarly. Since the number of 1's on each bitmap is less than the number of 0's, the bitmap can be stored using the sparse bitmap of [23].

The remaining operations require $\texttt{rank}$ and $\texttt{select}$ on P, or the virtual bit vectors P_1 and P_2. For $\texttt{rank}$, it is necessary to store the answers at the end of each block, finishing the query inside the corresponding block. For $\texttt{select}_1$ (and $\texttt{select}_0$), we build a sequence with the accumulated number of 1's in each of the τ blocks of P. Such a sequence is stored in a bitmap, representing each number in unary as gaps of 0's between consecutive 1's using the results of [23].

2.1.3. Memory space

To analyze the space used for the simplified NS-representation, observe that storing P requires $2n$ bits, while the space needed to store the vectors e', m', M', and n' is $2(n/s)\lg n = 2n/w$. The space needed to store the same vectors for the RMMT of P_1 is the same. Since we can assume that $w = \Omega(\lg n)$, the total size of the simplified RMMT is thus $2n + O(n/\lg n)$ bits.

The NS-representation that supports constant time queries requires the construction of $\tau = \lceil 2n/w^c \rceil$ RMMTs over sequences of w^c parentheses. Thus, the τ RMMTs require $2n + O(n/\lg n)$ bits. The additional data structures needed to support constant time queries add some extra space: To support $\texttt{fwd\_search}$, the ladders use $O(\frac{n \lg n}{w^c})$ bits, the arrays of ancestors use $O(\frac{n \lg^2 n}{w^c})$ bits, the sparse bitmap uses $O(\frac{n \lg w^c}{w^c} + \frac{nt^t}{\lg^t n} + n^{3/4})$ bits and the fusion trees use $O(\frac{n \lg^2 n}{w^c})$ bits. Thus, the extra structures to support $\texttt{fwd\_search}$ use $O(\frac{n \lg^2 n}{w^c} + \frac{nt^t}{\lg^t n} + n^{3/4})$ bits, with $t > 0$. The $\texttt{rmqi}$ and $\texttt{RMQi}$ query structures add $O(n/w^c)$ extra bits. Since there are $O(n/w^c)$ marked blocks, the $\texttt{degree}$ operation uses $O(n \lg n/w^c)$ extra bits. The bitmaps of the remaining operations, such as $\texttt{child}$ and $\texttt{child\_rank}$, uses $\frac{2n}{w^c}\lg(w^c) + O(\frac{nt^t}{\lg^t n} + n^{3/4})$ extra bits, supporting rank/select operations in $O(t)$ time, since they correspond to the sparse bitmap of [23]. Therefore, the total space used by the additional data structures is $O(\frac{n(c \lg w + \lg^2 n)}{w^c} + \frac{nt^t}{\lg^t n} + n^{3/4} + \sqrt{2^w})$ bits, where the term $\sqrt{2^w}$ corresponds to the lookup tables. With $w = \Omega(\lg n)$ and $t = c$, the extra space is $O(\frac{n(c \lg \lg n + \lg^2 n) + nc^c}{\lg^c n} + n^{3/4} + \sqrt{n})$ bits. Combined with the $2n + O(n/\lg n)$ bits of the RMMTs, the NS-representation requires $2n + O(n/\lg n + \frac{n(c \lg \lg n + \lg^2 n)}{\lg^c n})$ bits, with $c > 0$, supporting queries in $O(c)$ time.

According to [2], the $O(n/\lg n)$ space of the RMMTs can be reduced by using *aB-trees* [23]. Given an array A of size N, with N a power of B, an aB-tree is a complete tree of arity B, that stores B consecutive elements of A on its leaves. Besides, each node of the aB-tree stores a value $\varphi \in \Phi$. For the leaves, φ must be a function of the elements of A that it stores; for

```
Input: A, v, s, e
1 c := 0
2 if e − s = 1 then
3 |   if A[s] = v then return 1
4 |   return 0
5 m := ⌊(s + e)/2⌋
6 a := spawn pcount(A, v, s, m)
7 b := pcount(A, v, m + 1, e)
8 sync
9 return a + b
```

Algorithm 1: pcount(). Example of a parallel recursive algorithm using the **spawn** and **sync** keywords. In parallel, the algorithm counts the occurrences of the element v between the s-th and e-th elements of the subarray A.

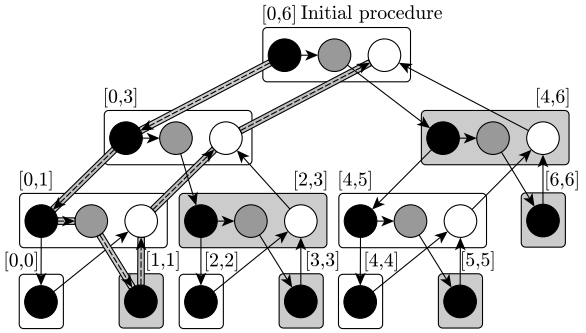


Fig. 4. Example of a multithreaded computation on the Dynamic Multithreading Model. It corresponds to the Directed Acyclic Graph representation of the call $pcount(A, v, 0, 6)$ of the Algorithm 1. Vertices represent strands and edges represent dependences.

internal nodes, φ must be a function of the φ-values of its children. An aB-tree can decode the B φ-values of the children of any internal node and the B values of A for the leaves in constant time, if they are packed in a machine word. An aB-tree can be stored in $N + 2 + O(\sqrt{2^w})$ bits (See [23] for the details). Thus, with $A = P$, $B = k = s = O(\frac{w}{c \lg w})$, φ-values encoding e', m', M', n' values and blocks of size $N = B^c$, it is possible to store each RMMT in $N + 2 + O(\sqrt{2^w})$ bits. The sum of all the RMMTs is $2n + O(\frac{n}{B^c} + \sqrt{2^w}) = 2n + O(\frac{n(c \lg \lg n)^c}{\lg^c n} + \sqrt{2^w})$. Finally, using aB-trees to store the RMMTs, the space usage of the NS-representation is reduced to $2n + O(\frac{n(c \lg \lg n + \lg^2 n)}{\lg^c n} + \sqrt{2^w})$ bits.

2.2. Dynamic multithreading (DyM) model

In the DyM model [27, Chapter 27], a *multithreaded computation* is modelled as a directed acyclic graph $G = (V, E)$ whose vertices are instructions and where $(u, v) \in E$ if u must be executed before v. The time T_p needed to execute the computation on p cores depends on two parameters of the computation: its *work* T_1 and its *span* T_∞. The work is the running time on a single core, that is, the number of nodes (i.e., instructions) in G, assuming each instruction takes constant time. Since p cores can execute only p instructions at a time, we have $T_p = \Omega(T_1/p)$. The span is the length of the longest path in G. Since the instructions on this path need to be executed in order, we also have $T_p = \Omega(T_\infty)$. Together, these two lower bounds give $T_p = \Omega(T_\infty + T_1/p)$. Work-stealing schedulers match the optimal bound to within a factor of 2 [28]. The degree to which an algorithm can take advantage of the presence of $p > 1$ cores is captured by its *speed-up* T_1/T_p and its *parallelism* T_1/T_∞. In the absence of cache effects, the best possible speed-up is p, known as *linear speed-up*. Parallelism provides an upper bound on the achievable speed-up.

To describe parallel algorithms in the DyM model, we augment sequential pseudocode with three keywords. The **spawn** keyword, followed by a procedure call, indicates that the procedure should run in its own thread and *may* thus be executed in parallel to the thread that spawned it. The **sync** keyword indicates that the current thread must wait for the termination of all threads it has spawned. It thus provides a simple barrier-style synchronization mechanism. Finally, **parfor** is "syntactic sugar" for **spawn**ing one thread per iteration in a for loop, thereby allowing these iterations to run in parallel, followed by a **sync** operation that waits for all iterations to complete. In practice, the **parfor** keyword is implemented by halving the range of loop iterations, **spawn**ing one half and using the current procedure to process the other half recursively until reaching one iteration per range. After that, the iterations are executed in parallel. This implementation adds an overhead to the parallel algorithm bounded above by the logarithm of the number of loop iterations. When a procedure exits, it implicitly performs a **sync** to ensure all threads it spawned finish first. If a stream of instructions does not contain one of the above keywords, or a **return** (which implicitly **sync**'s) from a procedure, we group these instructions into a single *strand*.

For example, Algorithm 1 represents a parallel algorithm using **spawn** and **sync**, and Fig. 4 shows its multithreaded computation. In the figure, each circle represents one strand and each rounded rectangle represents strands that belong to

Input : An adjacency list representation of T consisting of arrays V and E and the number of threads, *threads*.
Output: The balanced parenthesis sequence P of T.

```
1  ET := an array of length 2|E|
2  P := an array of length 2|E| + 2
3  chk := |E|/threads
4  parfor t := 0 to threads − 1 do
5      for i := 0 to chk − 1 do
6          j := t ∗ chk + i
7          ET[2 ∗ j].value := 1 // forward edge, opening parenthesis
8          ET[2 ∗ j + 1].value := 0 // backward edge, closing parenthesis
9          if E[j].child is a leaf then
10             ET[2 ∗ j].succ := 2 ∗ j + 1
11         else
12             ET[2 ∗ j].succ := 2 ∗ next(E[j].child)
13         if E[j] is the last edge in the adjacency list of E[j].parent then
14             ET[2 ∗ j + 1].succ := 2 ∗ first(E[j].parent) + 1
15         else
16             ET[2 ∗ j + 1].succ := 2 ∗ next(E[j].parent)
17 parallel_list_ranking(ET)
18 parfor t := 0 to threads − 1 do
19     for i := 0 to 2 ∗ chk − 1 do
20         P[ET[2 ∗ t ∗ chk + i + 1].rank] := ET[2 ∗ t ∗ chk + i + 1].value
21 P[0] := 1
22 P[2|E| + 1] := 0
```

Algorithm 2: PFEA.

the same procedure call. Let $A[s, e]$ be the subarray with elements $A[s], A[[s + 1], \ldots, A[e]$. The algorithm starts on the initial procedure call with the subarray $A[0, 6]$. The first half of the subarray is spawned (black circle in the initial call) and the second half is processed by the same procedure (gray circle of the initial call). This divide-and-conquer strategy is repeated until reaching strands with one element of the array A (black circles on the bottom of the figure, where s is equal to e). Once a bottom strand is finished, it syncs to its calling procedure (white circles), until reaching the final strand (white circle of the initial call). Assuming that each strand takes unit time, the work is 25 time units and the span is 8 time units (this is represented in the figure by the nodes connected with shaded edges). For more examples of the usage of the DYM model, see [27, Chapter 27].

3. A parallel algorithm for succinct tree construction

In this section, we describe our new parallel algorithm for constructing the RMMT of a given tree, called the *Parallel Succinct Tree Algorithm* (PSTA). Its input is the balanced parenthesis sequence P of an n-node tree T. This is a tree representation commonly used in practice, particularly in secondary storage. For trees whose folklore encoding is not directly available, in Section 3.1 we describe a parallel algorithm that can compute such an encoding in $O(n/p + \lg p)$ time. Our algorithms assume that manipulating w bits takes constant time. Additionally, we assume the (time and space) overhead of scheduling threads on cores is negligible. This is guaranteed by the results of [28], and the number of available processing units in current systems is generally much smaller than the input size n, so this cost is indeed negligible in practice.

3.1. Parallel folklore encoding algorithm

The PSTA algorithm requires the balanced parentheses representation P of the input tree T, but in some applications T may not be given in this form. Here, we present a parallel algorithm that constructs the balanced parenthesis sequence of T from a representation of T stored in adjacency list representation. Since the balanced parenthesis sequence of T is also known as its *folklore encoding*, we call the algorithm the *Parallel Folklore Encoding Algorithm* (PFEA). The input tree is represented by an array of nodes, V, and an array of edges, E. Each node v in V stores a pointer to an adjacency list with one entry per edge incident to v, sorted counterclockwise around v, starting with v's parent edge. Each entry in this adjacency list points to v and to the edge in E it represents. Each edge $e = (u, v)$ in E points to its corresponding entries in the adjacency lists of u and v. Edges are assumed to be directed from parents to children. Thus, for an edge $e = (u, v)$, we refer to u and v as $e.parent$ and $e.child$, respectively. For $x \in \{u, v\}$, we use $next(e.x)$ to denote the index in E of e's successor element in x's adjacency list, and $first(v)$ to denote the index in E of the first element in v's adjacency list. Both are easily computed in constant time by following pointers.

The idea behind the construction is the following: Given an Euler tour of T that visits the children of each node in left-to-right order, the balanced parenthesis representation of T can be obtained by following the Euler tour, writing down an opening parenthesis for every edge traversed from parent to child and a closing parenthesis for every edge traversed from child to parent, and finally enclosing the resulting sequence in a pair of parentheses representing the root of T.

Algorithm 2 shows the pseudo-code of the construction. It creates two arrays, one an auxiliary array ET of length $2|E|$ to store the Euler tour of T, the other an array P of size $2|E| + 2$ to store the balanced parenthesis representation of T

Input : P, s, *threads*.
Output: RMMT represented as arrays e', m', M', n' and universal lookup tables.

```
1  o := ⌈2n/s⌉ − 1 // # internal nodes
2  e' := array of size ⌈2n/s⌉
3  m', M', n' := arrays of size ⌈2n/s⌉ + o
4  ct := ⌈2n/s⌉/threads
5  parfor t := 0 to threads − 1 do
6      e'_t, m'_t, M'_t, n'_t := 0
7      for chk := 0 to ct − 1 do
8          low := (t ∗ ct + chk) ∗ s
9          up := low + s
10         for par := low to up − 1 do
11             e'_t += 2 ∗ P[par] − 1
12             if e'_t < m'_t then
13                 m'_t := e'_t; n'_t := 1
14             else if e'_t = m'_t then
15                 n'_t += 1
16             else if e'_t > M'_t then
17                 M'_t := e'_t
18         e'[t ∗ ct + chk] := e'_t
19         m'[t ∗ ct + chk + o] := m'_t
20         M'[t ∗ ct + chk + o] := M'_t
21         n'[t ∗ ct + chk + o] := n'_t
22 parallel_prefix_sum(e', ct)
23 parfor t := 1 to threads − 1 do
24     for chk := 0 to ct − 1 do
25         if chk < ct − 1 then
26             e'[t ∗ ct + chk] += e'[t ∗ ct − 1]
27         m'[t ∗ ct + chk + o] += e'[t ∗ ct − 1]
28         M'[t ∗ ct + chk + o] += e'[t ∗ ct − 1]
```

Algorithm 3: PSTA (part I).

```
1  lvl := ⌈lg threads⌉
2  parfor st := 0 to 2^lvl − 1 do
3      for l := ⌈lg(2n/s)⌉ − 1 downto lvl do
4          for d := 0 to 2^(l−lvl) − 1 do
5              i := d + 2^l − 1 + st ∗ 2^(l−lvl)
6              concat(i, m', M', n')
7  for l := lvl − 1 downto 0 do
8      parfor d := 0 to 2^l − 1 do
9          i := d + 2^l − 1
10         concat(i, m', M', n')
```

Algorithm 4: PSTA (part II).

(lines 1–2). Each entry in *ET* represents the traversal of an edge of T and stores three values: *value* is "(" or ")" depending on whether the edge is traversed from parent to child or from child to parent, that is, it's the corresponding parenthesis to be added to P; *succ* is the index in *ET* of the next edge in the Euler tour; and *rank* is the rank in the Euler tour. Lines 4–16 of the algorithm populate *ET* with entries representing the Euler tour but leaving the *rank* values uninitialized. Line 17 computes ranks using a parallel list ranking algorithm [29]. Given these ranks, the balanced parenthesis representation can be obtained by writing $ET[i].value$ into $P[ET[i].rank]$. Lines 18–22 do exactly this.

3.2. Parallel succinct tree algorithm

Before describing the PSTA algorithm, we observe that the entries in e' corresponding to internal nodes of the RMMT need not be stored explicitly. This is because the entry of e' corresponding to an internal node is equal to the entry that corresponds to the last leaf descendant of this node; since the RMMT is complete, we can easily locate this leaf in constant time. Thus, our algorithm treats e' as an array of length $\lceil 2n/s \rceil$ with one entry per leaf. Our algorithm consists of three phases. In the first phase (Algorithm 3), it computes the leaves of the RMMT, i.e., the array e', as well as the entries of m', M' and n' that correspond to leaves. In the second phase (Algorithm 4), the algorithm computes the entries of m', M' and n' corresponding to internal nodes of the RMMT. In the third phase (Algorithm 5), it computes the universal lookup tables used to answer queries. The input to our algorithm consists of the balanced parenthesis sequence, P, the size of each chunk, s, and the number of available threads, *threads*.

To compute the entries of arrays e', m', M', and n' corresponding to the leaves of the RMMT (Algorithm 3), we first assign the same number of consecutive chunks, *ct*, to each thread (line 4). We call such a concatenation of chunks assigned to a

```
1  parfor x := −w to w − 1 do
2      parfor y := 0 to √(2^w) − 1 do
3          i := ((x + w) << w) OR w
4          near_fwd_pos[i] := w
5          p, excess := 0
6          repeat
7              excess += 1 − 2 ∗ ((y AND(1 << p)) = 0)
8              if excess = x then
9                  near_fwd_pos[i] := p
10                 break
11             p += 1
12         until p ≥ w
```

Algorithm 5: PSTA (part III).

```
Input: i, m′, M′, n′.
1  m′[i] := min(m′[2i + 1], m′[2i + 2])
2  M′[i] := max(M′[2i + 1], M′[2i + 2])
3  if m′[2i + 1] < m′[2i + 2] then
4      n′[i] := n′[2i + 1]
5  else if m′[2i + 1] > m′[2i + 2] then
6      n′[i] := n′[2i + 2]
7  else if m′[2i + 1] = m′[2i + 2] then
8      n′[i] := n′[2i + 1] + n′[2i + 2]
```

Function concat.

single thread a *superchunk*. For simplicity, we assume that the total number of chunks, $\lceil 2n/s \rceil$, is divisible by *threads*. Each thread then computes the *local* excess value of the last position in each of its assigned chunks, as well as the minimum and maximum local excess in each chunk, and the number of times the minimum local excess occurs in each chunk (lines 8–17). These values are stored in the entries of e', m', M', and n' corresponding to this chunk (lines 18–21). The local excess value of a position i in P is defined to be $\texttt{sum}(P, \pi, j, i)$, where j is the index of the first position of the superchunk containing position i. Note that the locations with minimum local excess in each chunk are the same as the positions with minimum global excess because the difference between local and global excess is exactly $\texttt{sum}(P, \pi, 0, j-1)$. Thus, the entries in n' corresponding to leaves store their final values at the end of the loop in lines 5–21, while the corresponding entries of e', m', and M' store *local* excess values.

To convert the entries in e' into global excess values, observe that the global excess at the end of each superchunk equals the sum of the local excess values at the ends of all superchunks up to and including this superchunk. Thus, we use a parallel prefix sum algorithm [29] in line 22 to compute the global excess values at the ends of all superchunks and store these values in the corresponding entries of e'. The remaining local excess values in e', m', and M' can now be converted into global excess values by increasing each by the global excess at the end of the preceding superchunk. Lines 23–28 do exactly this.

The computation of entries of m', M', and n' (Algorithm 4) first chooses the level closest to the root that contains at least *threads* nodes and creates one thread for each such node v. The thread associated with node v calculates the m', M', and n' values of all nodes in the subtree rooted at v, by applying the function *concat* to the nodes in the subtree bottom up (lines 2–6). The invocation of this function for a node computes its m', M', and n' values from the corresponding values of its children. With a scheduler that balances the work, such as a work-stealing scheduler, cores have a similar workload. Lines 7–10 apply a similar bottom-up strategy for computing the m', M', and n' values of the nodes in the top *lvl* levels, but they do this by processing these levels sequentially and, for each level, processing the nodes on this level in parallel.

Algorithm 5 illustrates the construction of universal lookup tables using the construction of the table *near_fwd_pos* as an example. This table is used to support the `fwd_search` operation (see Section 2.1). Other lookup tables can be constructed analogously. As each entry in such a universal table can be computed independently, we can easily compute them in parallel.

3.3. Theoretical analysis

In the PFEA algorithm, lines 4–16 and 18–22 perform $O(n)$ work and have $T_p = O(n/p + \lg p)$ and span $T_\infty = O(\lg n)$. The $O(\lg p)$ and $O(\lg n)$ terms in T_p and in the span correspond to the implicit overhead of the parallel loop in line 4. The whole computation here (and in lines 18–22) could have been formulated as a single parallel loop. However, in the interest of limiting scheduling overhead, we create only as many parallel threads as necessary, similar to the PSTA algorithm in Section 3.2. Line 17 performs $O(n)$ work and has $T_p = O(n/p + \lg p)$ and span $O(\lg n)$. This gives a total work of $T_1 = O(n)$ and a span of $T_\infty = O(\lg n)$. The running time on p cores is $T_p = O(n/p + \lg p)$.

The analysis of the PSTA algorithm is done in three steps: Lines 1–21 of Algorithm 3 require $O(n)$ work, $T_p = O(n/p + \lg p)$ and span $T_\infty = O(\lg n)$. Line 22 requires $O(p)$ work, $T_p = O(\lg p)$ and span $T_\infty = O(\lg n)$, because we compute a prefix sum over only p values. Lines 23–28 require $O(n)$ work, $T_p = O(n/p + \lg p)$ and span $T_\infty = O(\lg n)$. Lines 1–6

of Algorithm 4 require $O(n/s)$ work, $T_p = O(n/sp)$ and span $T_\infty = O(\lg n/s)$. Lines 7–10 require $O(p)$ work, $T_p = O(\lg p)$ and span $T_\infty = O(\lg n/s)$. Building the top lvl levels of the RMMT can be reduced to the prefix sum problem, since minimum/maximum operations are associative. Algorithm 5 requires $O(\sqrt{2^w}\,\mathrm{poly}(w))$ work, $T_p = O(\sqrt{2^w}\,\mathrm{poly}(w)/p)$ and has span $O(\lg\sqrt{2^w} + \lg\mathrm{poly}(w))$. As was defined in Section 2.1, w is the machine word size. Thus, the total work of PSTA is $T_1 = O(n + \lg p + \sqrt{2^w}\,\mathrm{poly}(w))$ and its span is $O(\lg n)$. This gives a running time of $T_p = O(T_1/p + T_\infty) = O(n/p + \lg p + \sqrt{2^w}\,\mathrm{poly}(w)/p)$ on p cores.[3] The speedup is $T_1/T_p = O\left(\frac{p(n+\sqrt{2^w}\,\mathrm{poly}(w))}{n+\sqrt{2^w}\,\mathrm{poly}(w)+p\lg p}\right)$. Under the assumption that $p \ll n$, the speedup approaches $O(p)$. Moreover, the parallelism T_1/T_∞ (the maximum theoretical speedup) of PSTA is $\frac{n+\sqrt{2^w}\,\mathrm{poly}(w)}{\lg n}$.

The PSTA algorithm does not need any extra memory related to the use of threads. Indeed, it just needs space proportional to the input size and the space needed to schedule the threads. A work-stealing scheduler, like the one used by the DyM model, exhibits at most a linear expansion space, that is, $O(S_1 p)$, where S_1 is the minimum amount of space used by the scheduler for any execution of a multithreaded computation using one core. This upper bound is optimal within a constant factor [28]. In summary, the working space needed by our algorithm is $O(n\lg n + S_1 p)$ bits. Since in our algorithm the scheduler does not need to consider the input size to schedule threads, $S_1 = O(1)$. Thus, since in modern machines it is usual that $p \ll n$, the scheduling space is negligible and the working space is dominated by $O(n\lg n)$.

Note that in succinct data structure design, it is common to adopt the assumption that $w = \Theta(\lg n)$, and when constructing lookup tables, consider all possible bit vectors of length $(\lg n)/2$ (instead of $w/2$). This guarantees that the universal lookup tables occupy only $o(n)$ bits. Adopting the same strategy, we can simplify our analysis and obtain $T_p = O(n/p + \lg p)$.

Thus, we have the following theorem:

Theorem 1. *The balanced parenthesis sequence representation of an ordinal tree on n nodes can be computed with $O(n)$ work, $O(\lg n)$ span and $O(n\lg n)$ bits of working space. Given a balanced parenthesis sequence of an ordinal tree on n nodes, a $(2n + O(n/\lg n))$-bit representation can be computed with $O(n)$ work, $O(\lg n)$ span and $O(n\lg n)$ bits of working space. This compact representation supports the operations in Table 1 in $O(\lg n)$ time.*

3.4. Parallel algorithm to support constant-time queries

In this section we show how to construct the *2d-Min-Heap* and its *ladders*, the *sparse bitmap* of Pătraşcu, *fusion trees* and *range-minimum-query* structure in parallel, plus the computation of marked blocks. All of these structures are built over the minima, maxima, excess and the number of minima values of the $\tau = \lceil 2n/w^c \rceil$ RMMTs and are used to support different operations over trees in constant time (see Section 2.1.2).

2d-min-heap and ladders. Let $S = (x_0, x_1, \ldots, x_n = -\infty)$ be a sequence on n integers. Let the *closest smaller successor* of x_i be the element x_j such that $j = min\{j' | j' > i \wedge x_{j'} < x_i\}$. Thus, x_j is the parent of x_i in the 2d-Min-Heap. The 2d-Min-Heap is then fully determined once we find the closest smaller successor of all elements $x_i \in S$.

Let C be the cartesian tree of S [30]. Let the *closest right ancestor* of x_i in C be the closest ancestor x_j of x_i such that x_i is in the left subtree of x_j. Since $x_n = -\infty$, both the closest smaller successor and the closest right ancestor are well-defined for all x_i, where $0 \le i \le n-1$. Observe that the closest smaller successor of x_i and the closest right ancestor are the same element.

Let $ET = (y_0, y_1, \ldots, y_m)$ be the Euler tour of C that visits the children of each node in right-to-left order. To ensure that each node in C has two children, we add (virtual) dummy nodes. We assume that every node x_i in C, and hence in ET, is labelled with its index i in S. We also assume that for some x_i in C, we know the first occurrence of x_i in ET. Both these assumptions can be guaranteed as part of the construction of ET. The dummy nodes of C are not represented explicitly in ET. Thus, for instance, a node x_i with two dummy nodes (a leaf of C) is represented by three consecutive instances of i. We can obtain the closest right ancestor by performing a list ranking of ET, computing the sequence $\delta(ET) = (z_0, z_1, \ldots, z_m)$ defined as $z_0 = y_0$ and $z_i = \delta(z_{i-1}, y_i)$, for all $1 \le i \le n$, where $\delta(\cdot,\cdot)$ is defined as

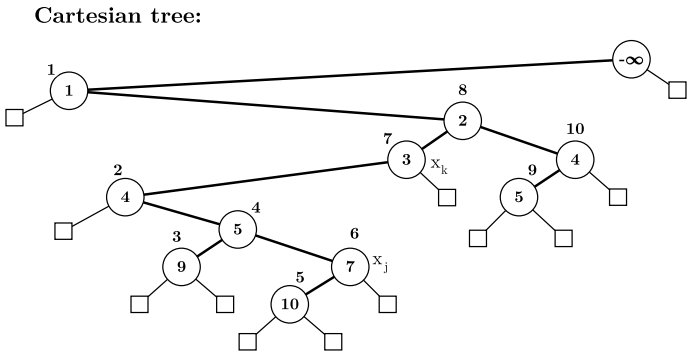
$$\delta(x, y_i) = \begin{cases} y_i, & \text{if } s(i) < i, \text{ where } s(i) \text{ denotes the index of } y_{i+1} \text{ element in } ET \\ x, & \text{otherwise} \end{cases}$$

See Fig. 5b as an example of the Euler Tour ET of the tree in Fig. 5a and its corresponding sequence $\delta(ET)$.

Lemma 1. *If y_i is the first occurrence of some element x_j in ET, then the element z_{i-1} in $\delta(ET)$ is x_j's closest right ancestor in C.*

Proof. Since ET visits all the descendants of a node x_k after the first occurrence of x_k in ET, the first occurrence of x_j in ET comes after the first occurrence of x_k if x_k is x_j's closest right ancestor. Now assume that y_h is the last occurrence of x_k before the first occurrence, y_i, of x_j. Let $(y_h, y_{h+1}, \ldots, y_i)$ be the subsequence of ET between y_h and y_i, and let

[3] Notice that the term $\lg n$ of the span is implicit in the term $n/p + \lg p$ of T_p. When $p \le n/\lg n \to n/p \ge \lg n$. When $p > n/\lg n \to \lg p = \Theta(\lg n)$.



(a) Cartesian tree of the sequence $(1, 4, 9, 5, 10, 7, 3, 2, 5, 4, -\infty)$. Dummy nodes are represented with squares.

indexes : 11 1 8 10 10 9 9 9 10 8 7 7 2 4 6 6 5 5 5 6 4 3 3 3 4 2 2 7 8 1 1 11

ET : -∞ 1 2 4 4 5 5 5 4 2 3 3 4 5 7 7 10 10 10 7 5 9 9 9 5 4 4 3 2 1 1 -∞

δ(ET) : -∞ -∞ -∞ -∞ 4 4 4 4 4 2 2 3 3 3 3 7 7 7 7 7 5 5 5 5 5 5 5 2 2 2

(b) Euler Tour ET of the cartesian tree in Fig. 5a and the sequence $\delta(ET)$. The first ocurrence of each node of the cartesian tree is highlighted with a square. The arrows show the closest right ancestor of each node.

Fig. 5. Example of the proof of Lemma 1. The resulting 2d-Min-Heap corresponds to the tree in Fig. 3a.

$(k = j_h, j_{h+1}, \ldots, j_i = j)$ be the sequence of indices of S such that $y_t = x_{j_t}$, for all $h \le t \le i$, where x_{j_t} is the j_t-th element in ET.

To prove the lemma, we need to show that $j_{h+1} < j_h$ and $j_{t+1} > j_t$ for all $h < t < i$ because this implies that $\delta(x, y_h) = y_h = x_k$ and $\delta(x, y_t) = x$ for all $h < t < i$, that is, $z_h = \delta(z_{h-1}, y_h) = x_k$ and $\delta(z_{t-1}, y_t) = z_{t-1} = z_h = x_k$ for all $h < t \le i$; in particular, $z_{i-1} = x_k$, as claimed. The node y_{h+1} must be x_k's left child in C because the last visit to x_k by ET before visiting any node in x_k's left subtree happens immediately before visiting x_k's left child. Thus, $j_{h+1} < j_h$. All the nodes on the path from y_{h+1} to x_j in C are left ancestors of x_j because x_k is the closest right ancestor of x_j. Thus, by the definition of ET, $(y_{h+1}, y_{h+2}, \ldots, y_i)$ is the sequence of nodes in this path. Since y_{t+1} is the right child of y_t for all $h < t < i$, we have that $j_{t+1} > j_t$.

See Fig. 5 as an illustration of this proof. In the figure, let us take x_k as the element with value 3 and index 7 and x_j as the element with value 7 and index 6. In ET, the first occurrence of the element 7 is at position $i = 14$ and the last occurrence of 3 before position 14 is at position $h = 11$. Thus, for this example, $(y_h, y_{h+1}, \ldots, y_i) = (3, 4, 5, 7)$ and $(j_h, j_{h+1}, \ldots, j_i) = (7, 2, 4, 6)$. □

We can parallelize this strategy using the results of [31] to obtain the cartesian tree C of S with $O(n)$ work, $O(\lg^2 n)$ span and $O(n)$ working space, our PFEA algorithm in Section 3.1 to compute the Euler tour ET with $O(n)$ work, $O(\lg n)$ span and $O(n \lg n)$ working space, and the results of [29] to compute the list ranking using the function $\delta(\cdot, \cdot)$ with $O(n)$ work, $O(\lg n)$ span and $O(n)$ working space.

Next we compute $\delta(ET)$ using list ranking (Lemma 1). Then, we assign one core to every element $z_i \in \delta(ET)$. The core writes z_i as the closest right ancestor of x_j if and only if y_i is the first occurrence of x_j in ET. This is done in $O(1)$ time.

Thus, the complexity of constructing the 2d-Min-Heap in parallel is $O(n)$ work, $O(\lg^2 n)$ span and $O(n \lg n)$ working space.

After constructing the 2d-Min-Heap, the tree is decomposed into ladders. The ladders are constructed by recursively extracting the longest path of the tree. This gives us a set of paths. Then, each path of length l is extended by adding at most l nodes towards the root. These extended paths are called ladders. To construct the ladders in parallel, assume that we have a tree $T^{\mathcal{B}}$ with a particular embedding $\mathcal{B}$: for each node v of $T^{\mathcal{B}}$, the children of v are ordered by their height, with the highest child in the leftmost position.

Lemma 2. *Given a tree $T^{\mathcal{B}}$ with embedding $\mathcal{B}$ and n nodes, the ladders of $T^{\mathcal{B}}$ can be constructed in parallel with $O(n \lg n)$ work, $O(\lg n)$ span and $O(n \lg n)$ working space.*

Proof. To prove the lemma, we need to compute the depth of each node of the tree. This can be done in parallel by using the PFEA algorithm of Section 3.1, adding 1 for each forward edge and subtracting 1 for each backward edge. It takes $O(n)$ work, $O(\lg n)$ span and $O(n \lg n)$ working space. Now, let $ET_n = (v_0, \ldots, v_m)$ be the Euler tour of $T^{\mathcal{B}}$ that visits the children of each node in left-to-right order and writes the index of each node. Let $ET_d = (v'_0, \ldots, v'_m)$ be the Euler tour

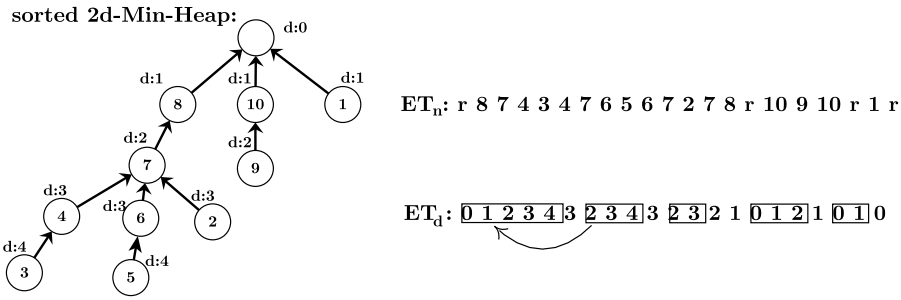


Fig. 6. Computation of the ladders of a tree $T^{\mathcal{B}}$, with embedding $\mathcal{B}$. The tree $T^{\mathcal{B}}$ is the result of applying the embedding $\mathcal{B}$ to the tree of Fig. 3. In the tree, the depth of each node is shown. For example, $d:3$ means that the depth of a node is 3. In the Euler Tour ET_n, the dummy root is represented by the symbol **r**.

of $T^{\mathcal{B}}$ that visits the children of each node in left-to-right order and writes the depth of each node. See Fig. 6 for an example. We can decompose the tree into paths by finding contiguous increasing subsequences in ET_d. By the definition of the embedding $\mathcal{B}$, the resulting paths are the same ones obtained by recursively extracting the longest path of the tree. For the path represented by the subsequence $ET_d[a..b]$, $ET_d[b]$ corresponds to the depth of the leaf of this path and the length of $ET_d[a..b]$ is $b-a+1$.

To compute the ladders, we need to extend each subsequence in ET_d. For a subsequence $ET_d[a..b]$, if $ET_n[a]$ is the root of the tree, then the subsequence does not need to be extended. Otherwise, the subsequence is extended by adding up to $x = (b-a-1)$ extra nodes. If $x=0$, then the subsequence does not need to be extended. The extra nodes that we need to add correspond to ancestors of the leaf $ET_n[b]$ at depths $ET_d[a]-i$, $i \in (1,\ldots,x)$. We use the operation `level_anc`$(ET_n[a], i)$, $i \in (1,\ldots,x)$, of the simplified NS-representation (see Table 1, operation 5) to obtain all the ancestors and the ladders.

The Euler Tours ET_n and ET_d can be found using the `PFEA` algorithm. The bounds of all increasing subsequences can be found in parallel by finding each index i, such that $ET_d[i] < ET_d[i-1]$ or $ET_d[i] > ET_d[i+1]$. This can be done with $O(n)$ work, $O(1)$ span and $O(n \lg n)$ working space. The simplified NS-representation can be constructed with $O(n)$ work, $O(\lg n)$ span and $O(n \lg n)$ working space. The `level_anc` operation of the simplified NS-representation takes $O(\lg n)$ time to be answered (operation 5 in Table 1). Since the total length of all the ladders is $2n$ [2], the amount of operations that we need to perform is $O(n)$. We can perform all the operations independently, so the $O(n)$ `level_anc` operations can be answered in parallel with $O(n \lg n)$ work and $O(\lg n)$ span. □

Alternatively, observe that we can extend a subsequence $ET_d[a..b]$ using wavelet trees [32–34]. Since the extra nodes that we need to add correspond to the ancestors of the leaf $ET_n[b]$, they appear before $ET_n[b]$ in the Euler Tours, with depths $ET_d[a]-i$, $i \in (1,\ldots,x)$. Given a node v at position j in ET_d, we know that the parent of v is at position k in ET_d, where $k = max\{k' | k' < j, ET_d[k'] = ET_d[j]-1\}$. In general, to extend the subsequence $ET_d[a..b]$, we need the nodes at positions $max\{k' | k' < a, ET_d[k'] = ET_d[a]-i\}$, with $i \in (1,\ldots,x)$. These positions can be found by using rank/select operations over ET_d. To answer the rank/select operations efficiently, we could construct a wavelet tree over ET_d, considering the contiguous alphabet $\Sigma = \{0, \lceil \lg |ET_d| \rceil - 1\}$. For example, to extend the subsequence $ET_d[a..b]$ with a node with depth d', we need to perform $\texttt{select}_{d'}(ET_d, \texttt{rank}_{d'}(ET_d, a))$. Finally, once we find the position of all the nodes, we use ET_n to obtain their indexes. To construct the wavelet trees in parallel, we can use Algorithm `PWT` or algorithm `DD` of [1]. With Algorithm `PWT`, the wavelet tree can be computed with $O(n \lg n)$ work, $O(n)$ span and $O(n \lg n)$ working space; with Algorithm `DD`, the wavelet tree can be computed with $O(n \lg n)$ work, $O(\lg n)$ span and $O(n^2 \lg n)$ working space. If there are $p < \lg n$ available threads, the working space of Algorithm `DD` is reduced to $O(n \lg^2 n)$. The rank/select operations over the wavelet tree take $O(\lg n)$ time. We can perform all the operations independently, so the $O(n)$ rank/select operations can be answered in parallel in $O(n \lg n)$ work and $O(\lg n)$ span.

The following lemma presents an extension of the Lemma 2 for trees with arbitrary embeddings.

Lemma 3. *Given a tree $T^{\mathcal{A}}$ with an arbitrary embedding $\mathcal{A}$ and n nodes, the ladders of $T^{\mathcal{A}}$ can be constructed in parallel with $O(n \lg n)$ work, $O(\lg n)$ span and $O(n \lg n)$ working space.*

Proof. To prove this lemma, we need to map the embedding $\mathcal{A}$ of $T^{\mathcal{A}}$ to $\mathcal{B}$. To compute the embedding $\mathcal{B}$, we need to order all the children of the nodes of $T^{\mathcal{A}}$ by height with the highest in the leftmost position. To do this, we use the `PFEA` algorithm to compute the folklore encoding of T and then construct its simplified NS-representation to use the `height` operation to obtain the height of all the nodes. Both the folklore encoding and the simplified NS-representation can be computed with $O(n)$ work, $O(\lg n)$ span and $O(n \lg n)$ working space. The `height` operation takes $O(\lg n)$ and there are n operations, and therefore all the operations can be done with $O(n \lg n)$ work and $O(\lg n)$ span. After that, we can use a parallel stable sorting algorithm over the children of the nodes of $T^{\mathcal{A}}$. Raman [35] sorts an array of n integers each in the

domain $[1,\ldots,m]$, for $m = n^{O(1)}$, with $O(n \lg\lg m)$ work, $O(\lg n/\lg\lg n + \lg\lg m)$ span and $O(n \lg m)$ working space. In our case, the total number of children in T is $n-1$ or $2(n-1)$ by using bidirectional edges, and the height of any node is less than n. Therefore, we can sort the children of all the nodes of T with $O(n \lg\lg n)$ work, $O(\lg n/\lg\lg n)$ span and $O(n \lg n)$ space.

With the new embedding $\mathcal{B}$, we use Lemma 2 to finish the proof.

In the NS-representation, the 2d-Min-Heap has τ nodes, and therefore, the 2d-Min-Heap and its ladders can be computed with $O(\tau \lg \tau)$ work, $O(\lg^2 \tau)$ span and $O(\tau \lg \tau)$ working space. □

Considering the results of Bender and Farach-Colton to solve the *level ancestor problem* [14], we can use Lemma 3 to parallelize their solution. Their solution is based on the computation of ladders, pointers to ancestors and lookup tables of a rooted tree. Ladders can be computed using Lemma 3. The pointers and lookup tables can be computed by traversing the tree, using the parallel Euler Tour algorithm introduced in Algorithm 2.

Pătraşcu's bitmap. Navarro and Sadakane use the sparse bitmap of Pătraşcu [23] to represent a bitmap with 2τ 1's and $2\tau w^c$ 0's using $O(\tau \lg w^c + \frac{\tau w^c t^t}{\lg^t(\tau w^c)} + (\tau w^c)^{3/4})$ bits and supporting rank/select queries in $O(t)$ time. Pătraşcu demonstrated how to use recursion to achieve a better redundancy. Given a sparse bitmap A of size n, the succinct representation of A is constructed as follows: Choose $B \geq 2$ such that $B \lg B = \frac{\varepsilon \lg n}{t}$, and $r = B^t = (\frac{\lg n}{t})^{\Theta(t)}$. We first divide the bitmap A into n/r segments of size r. Each segment is stored in a *succinct aB-tree*. Each succinct aB-tree is constructed by dividing the bitmap into B independent segments. On each small segment, the author applies Lemma 3 of [23] recursively t times. In order to reduce the redundancy, on each application of the lemma, M memory bits are extracted from the values of the independent segments and stored, and the rest of the unextracted bits, called *spill*, are passed to the next iteration. Then, Lemma 5 of [23] is applied in each succinct aB-tree, storing the last spill and memory bits in the root of each aB-tree. For each segment of size r, the index in memory of the segment's memory bits are stored. Additionally, the number of ones in each segment are stored in a partial sums vector and a predecessor structure to support rank and select operations, respectively.

The parallel algorithm to construct the Pătraşcu's bitmap is similar to the parallel algorithm we used to construct the `RMMT` in Section 3.2. First, we construct the n/r succinct aB-trees in parallel. On each aB-tree, we divide the bitmap on B independent bitmaps of size r/B, similar to the `PSTA` algorithm. We apply Lemma 3 of [23] recursively on each small bitmap, t times. Then, we apply Lemma 5 of [23] in each succinct aB-tree, storing the final *spill* and memory bits in the root of each aB-tree. After that, all the n/r succinct aB-trees are built with $O(n)$ work and $O(t)$ span. The next step consists of storing the values of the root of each aB-tree. To support the rank operation, we compute in parallel the partial sum vector of these values with $O(n/r)$ work and $O(\lg(n/r))$ span using a parallel prefix sum algorithm. To support the select operation, we use a fusion tree. Below, we will explain how to construct a fusion tree in parallel with $O(n/r)$ work and $O(\lg\lg(n/r))$ span. With all these steps, Pătraşcu's sparse bitmap can be computed in parallel with $r = (\frac{\lg n}{t})^{\Theta(t)}$, in $O(n + \frac{nt^t}{\lg^t n})$ work, $O(t + \lg(\frac{nt^t}{\lg^t n}))$ span and $O(n)$ working space. In the context of succinct trees, the work is $O(\tau w^c + \frac{\tau w^c t^t}{\lg^t \tau w^c})$, the span is $O(t + \lg(\frac{\tau w^c t^t}{\lg^t(\tau w^c)}))$ and the working space is $O(\tau w^c)$.

Fusion tree. A fusion tree stores an array A of size n of w-bit integers, supporting predecessor/successor queries in $O(\lg_w n)$ time. A fusion tree is essentially a B-tree with branching factor $w^{1/5}$, and therefore, if we can construct a B-tree over the array A in parallel, we also obtain a parallel algorithm to construct fusion trees. In [36], Wang and Chen present a parallel algorithm to construct B-trees in $O(\lg\lg n)$ span, for a sorted list. The $\lg\tau$ values of the accumulated weights sequence used to answer `fwd_search` queries are always increasing. Therefore, we can apply the algorithm described in [36] to construct the corresponding B-tree. Henceforth, we will consider the array A as a sorted list of n keys. Although the algorithm of Wang and Chen is based on the EREW model, it can be applied in SMP systems without any modifications. If there are p available cores, the complexity of the algorithm is $O(n/p)$.

Given the sorted list A, the algorithm of Wang and Chen constructs an uniquely defined B-tree with branching factor m and the following properties:

- The B-tree has the minimal height $h = \lceil \lg_m(n+1) \rceil + 1$.
- The root owns $\lceil (n+1)/m^h - 1 \rceil$ keys.
- There exists an integer c, $1 < c \leq h+1$, such that all the nodes of the B-tree above the c-th level contain $m-1$ keys and all the non-leaf nodes below the c-th level contain $\lceil m/2 \rceil - 1$ keys.
- The leftmost leaf of the B-tree contains s keys, $\lceil m/2 \rceil \leq s \leq m-1$. The rest of the leaves may contain s or $s-1$ keys, but may not own more keys than the leaf node on its left.

With this well-defined B-tree, the parallel algorithm computes the position of each key of A in the B-tree. Each node of the B-tree is identified by its order in a BFS traversal. The details of how to assign a position to each key of A are shown in [36].

Once we have the B-tree, we apply the *sketch* algorithm [24] to compress the keys in each node of the B-tree. We apply the sketch algorithm in parallel in each node of the tree, in $O(1)$ span. Hence, the fusion tree can be computed with $O(n)$ work, $O(\lg\lg n)$ span and $O(n)$ working space.

In the NS-representation, to support `fwd_search` and `bwd_search` operations, τ fusion trees are constructed over τ sorted arrays of $O(\lg\tau)$ integers. Considering the previous parallel bounds, the τ fusion trees can be constructed with $O(\tau\lg\tau)$ work, $O(\lg\lg\tau)$ span and $O(\tau\lg\tau)$ working space.

Range-minimum-query. In [37], Fisher and Heun present a data structure to answer range minimum/maximum queries in constant-time using $O(n)$ bits over an array A of n elements. The array A is preprocessed by dividing it into $\lceil n/s\rceil$ blocks, $B_1,\ldots,B_{\lceil n/s\rceil}$, of size $s=\lceil\frac{\lg n}{4}\rceil$. A query from i to j, $\texttt{RMQ}(A,i,j)$, is divided into at most three subqueries: One *in-block* query over the block $B_{\lfloor i/s\rfloor}$, one *out-of-block* query over the blocks $B_{\lfloor i/s\rfloor+1},\ldots,B_{\lfloor j/s\rfloor-1}$ and one *in-block* query over the block $B_{\lfloor j/s\rfloor}$. If i and j belong to the same block, then only one in-block query is necessary. The in-block queries allow us to obtain the minimum/maximum element inside a block. On the other hand, out-of-block queries allow us to obtain a minimum/maximum element from consecutive blocks.

To answer in-block queries, authors use the fact that each block B_x can be represented by an unique *canonical cartesian tree* $C^{can}_{B_x}$. The canonical cartesian tree of B_x is a cartesian tree with a total order $\prec$ defined as follows: $B_x[i]\prec B_x[j]$ iff $B_x[i]<B_x[j]$, or $B_x[i]=B_x[j]$ and $i<j$. The idea is to precompute all the answers for all C_s possible canonical cartesian trees, where $C_s=\frac{1}{s+1}\binom{2s}{s}$ is the number of the rooted trees on s nodes. Thus, all the answers are stored in a table $P[1,C_s][1,s][1,s]$. The first dimension of the table P corresponds to a descriptor of the blocks of size s. For all $\lceil n/s\rceil$ blocks of A, their descriptors are stored in an array T, requiring $O(s)$ time to compute each one [37].

To answer out-of-block queries, the minimum/maximum element of each block is stored in an array $A'[1,n']$, where $n'=\lceil n/s\rceil$. The array A' is divided into $\lceil n'/s\rceil$ blocks, $B'_1,\ldots,B'_{\lceil n'/s\rceil}$. A `RMQ` query over A' is answered as before: one out-of-block query and two in-block queries. The in-block queries can be answered by computing the descriptor of each block of A', storing them in an array T' and reusing the lookup table P. To answer the out-of-block queries of A', a two-level storage scheme is used. s contiguous blocks of A' are grouped into a *superblock* consisting of $s'=s^2$ elements. We precompute all the answers in A' that cover at least one such superblock and store them into a table M. Similarly, we precompute all the answers in A' that cover at least one block, but not over a superblock and store them into a table M'. Thus, to find the minimum/maximum element inside a superblock, we need to use the table M twice. Summarizing, an out-of-block query of A can be decomposed into two in-block queries in A' (using T' and P), two out-of-block queries in A' (using M') and one out-of-superblock query in A' (using M).

Finally, the solution of Fisher and Heun has $O(n)$ construction time, $O(\lg^3 n)$ construction space (over the $O(n)$ space of the structure) and $O(1)$ query time.

Since the minimum/maximum operation is associative, we can use a domain decomposition strategy to parallelize the construction of the solution of Fisher and Heun. In a domain decomposition strategy, the input array is divided into subarrays, and then each subarray is processed in parallel. Then, all the processed subarrays are merged. Thus, we can obtain a parallel solution with $T_1=O(n)$ work, $T_\infty=O(\lg n)$ span and the same space complexity. The term $O(\lg n)$ is due to the traversal of the blocks of size $s=\lceil\frac{\lg n}{4}\rceil$, which is done sequentially.

In the context NS-representation, we need to answer queries over the root of the τ `RMMT`s. Therefore, to answer range minimum/maximum queries we can construct the solution in [37] with $O(\tau)$ work, $O(\lg\tau)$ span and $O(\tau+\lg^3\tau)$ working space.

Degree, child and childrank operations. To support `degree`, `child` and `child_rank`, we need to compute marked blocks. Remember that pioneers are the tightest matching pairs of parentheses (i,j), with $j=\texttt{find\_close}(i)$, such that i and j belong to different blocks. A marked block is a block that has the opening parenthesis of a pioneer (i,j) such that i and j do not belong to consecutive blocks. To compute such marked blocks, we need to apply the `find_close` operation over all the τ blocks. Since the `find_close` operation can be computed in constant time, all marked blocks can be computed with $O(\tau)$ work and $O(1)$ span. The `child` and `child_rank` operations additionally need to construct a sparse bitmap C for each marked block, which encodes the number of children of the marked block, in left-to-right order, as gaps of 0's between 1's. Therefore, to construct each bitmap it is enough to find the position of each 1 in the bitmap. To do so, we perform a parallel prefix sum over the blocks fully contained in a marked block. Let j be a marked block. The bitmap C_j of j can be constructed as follows: for each block j' fully contained in j, if j' has at least one child of j, we obtain the number of children of j'. If j' is not a marked block, then the number of children corresponds to $n_{j'}$ (the number of minima of block j'); if j' is a marked node, the number of children is 1. Notice that if j' is marked, then the blocks contained in j' do not have any children of j and they will not be considered for the rest of the computation of C_j. After that, we perform a parallel prefix sum over the blocks that do have some children of j, considering their left-to-right order. The result of the prefix sum corresponds to the position of all the 1's in C_j. The final step is to write, in parallel, all the 1's that they correspond to. Following the same idea, we can compute a bitmap C that represents the concatenation of all the bitmaps of the marked nodes. The parallel prefix sum is the most expensive step of this algorithm, and its work is $O(\tau)$, its span is $O(\lg\tau)$ and its working space is $O(\tau\lg w^c)$ bits, which is dominated by the array of size $O(\tau)$ used in the prefix sum, where each element uses $O(\lg w^c)$ bits.

Thus, with $w=\Theta(\lg n)$ we have the following theorem:

Table 2
Datasets used in the experiments.

Dataset	Number of nodes (n)	Depth
wiki	249,376,958	5
prot	335,360,503	26
dna	577,241,094	305
ctree	1,073,741,823	30
osm	2,337,888,180	3

Theorem 2. *A $(2n + O(n/\lg^c n))$-bit representation of an ordinal tree on n nodes and its balanced parenthesis sequence can be computed with $O(n + \frac{n}{\lg^c n}\lg(\frac{n}{\lg^c n}) + c^c)$ work, $O(c + \lg(\frac{nc^c}{\lg^c n}))$ span and $O(n \lg n)$ bits of working space. This representation supports the operations in Table 1 in $O(c)$ time, with $c > 0$.*

Proof. Each of the τ RMMTs can be constructed with $O(\lg^c n)$ work, $O(\lg \lg^c n)$ span and $O(\lg^c n \lg \lg^c n)$ bits of working space using Theorem 1. All the τ RMMTs can be constructed with $O(n)$ work, $O(\lg \lg^c n)$ span and $O(n \lg n)$ working space. Using the results of this section, with $t = c$, the additional data structures can be constructed with $O(n + \frac{n}{\lg^c n}\lg(\frac{n}{\lg^c n}) + c^c)$ work, $O(c + \lg(\frac{nc^c}{\lg^c n}))$ span and $O(n + \frac{n}{\lg^{c-1} n})$ working space. Thus, the total work is $O(n + \frac{n}{\lg^c n}\lg(\frac{n}{\lg^c n}) + c^c)$, the maximum span is $O(c + \lg(\frac{nc^c}{\lg^c n}))$ and the total working space is $O(n \lg n)$ bits. □

4. Experimental results

In this section we present the experimental results of the implementations of our algorithms PSTA and PFEA.

4.1. Experimental setup

We implemented the PSTA algorithm in C and compiled it using GCC 4.9 with optimization level -O2 and using the -ffast-math flag.[4] All parallel code was compiled using the GCC Cilk branch. The same flags were used to compile libcds [38] and sdsl [39], which were written in C++.

Table 2 shows the five input datasets that we used in our experiments. The first two datasets were suffix trees built out of the DNA and protein data from the Pizza & Chili corpus,[5] using code from http://www.daimi.au.dk/~mailund/suffix_tree.html. The next two datasets were XML trees built out of the Wikipedia[6] and OpenStreetMap[7] dumps. The final input dataset was a complete binary tree of depth 30.

The experiments were carried out on a machine with four 16-core AMD Opteron™ 6278 processors clocked at 2.4 GHz, with 64 KB of L1 cache per core, 2 MB of L2 cache shared between two cores, and 6 MB of L3 cache shared between 8 cores. The machine had 189 GB of DDR3 RAM, clocked at 1333 MHz.

Running times were measured using the high-resolution (nanosecond) C functions in <time.h>. Memory usage was measured using the tools provided by malloc_count [40].

4.2. Experimental results of the PSTA algorithm

To evaluate the performance of our PSTA algorithm, we compare it against libcds and sdsl, which are state-of-the-art implementations of the RMMT. Both assume that the input tree is given as a parenthesis sequence, as we do here. Our implementation of the PSTA algorithm deviates from the description in Section 3.2 in that we do not store the array n', since libcds and sdsl do not store it and that the prefix sum computation in line 22 of the algorithm is done sequentially in our implementation. This changes the running time to $O(n/p + p)$ but simplifies the implementation. Since $p \ll n/p$ for the input sizes we are interested in and the numbers of cores available on current multicore systems, the impact on the running time is insignificant. In our experiments, the chunk size s was fixed at 256.

4.2.1. Running time and speed-up

Table 3 shows the wall clock times achieved by psta, the sequential version of psta, called seq, libcds, and sdsl on different inputs. Each time is the median achieved over five non-consecutive runs, reflecting our assumption that slightly increased running times are the result of "noise" from external processes such as operating system and networking tasks. Fig. 7 shows the speed-up compared to the running times of seq, and Fig. 8 shows the speed-up compared to sdsl.

[4] The code and data needed to replicate our results are available at http://www.inf.udec.cl/~josefuentes/suctree.
[5] http://pizzachili.dcc.uchile.cl.
[6] http://dumps.wikimedia.org/enwiki/20150112/enwiki-20150112-pages-articles.xml.bz2 (January 12, 2015).
[7] http://wiki.openstreetmap.org/wiki/Planet.osm (January 10, 2015).

Table 3
Running times of `libcds`, `sdsl`, and `PSTA`, in seconds. `seq` corresponds to the sequential execution of `PSTA`.

p	wiki	prot	dna	ctree	osm
libcds	33.17	44.27	75.93	140.71	339.43
sdsl	1.94	2.67	4.57	8.35	18.10
seq	2.81	4.10	7.25	12.14	28.00
seq	2.81	4.10	7.25	12.14	28.00
1	2.81	4.10	7.15	12.17	28.05
4	.72	1.05	1.86	3.05	7.07
8	.40	.58	.95	1.57	3.55
12	.31	.43	.72	1.12	2.55
16	.24	.32	.55	.85	1.89
20	.19	.29	.49	.74	1.58
24	.19	.26	.42	.68	1.45
28	.16	.25	.43	.62	1.30
32	.18	.25	.38	.62	1.16
36	.20	.21	.36	.52	1.08
40	.21	.23	.35	.50	1.04
44	.22	.25	.34	.51	.97
48	.21	.26	.37	.49	.99
52	.27	.30	.36	.50	.93
56	.30	.36	.42	.50	.93
60	.27	.40	.39	.50	.93
64	.30	.33	.38	.54	.90

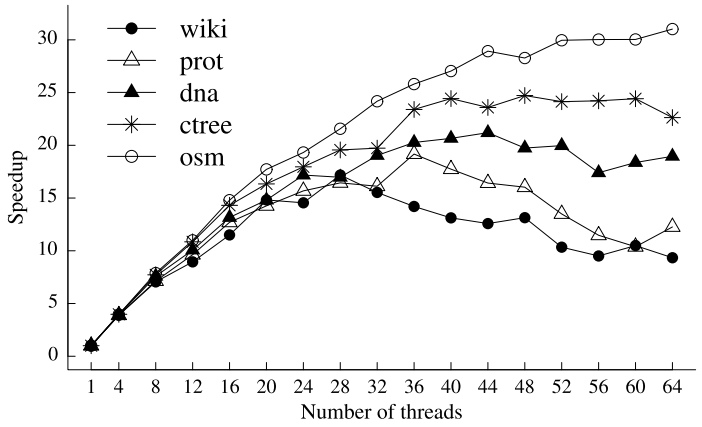


Fig. 7. Speed-up of `PSTA` compared to `seq`.

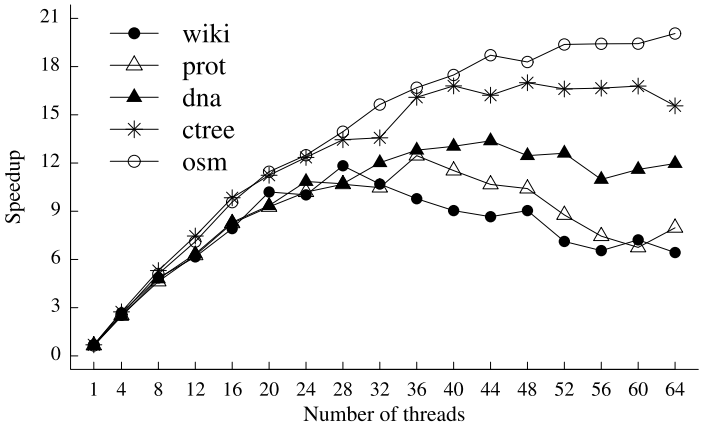


Fig. 8. Speed-up of `PSTA` compared to `sdsl`.

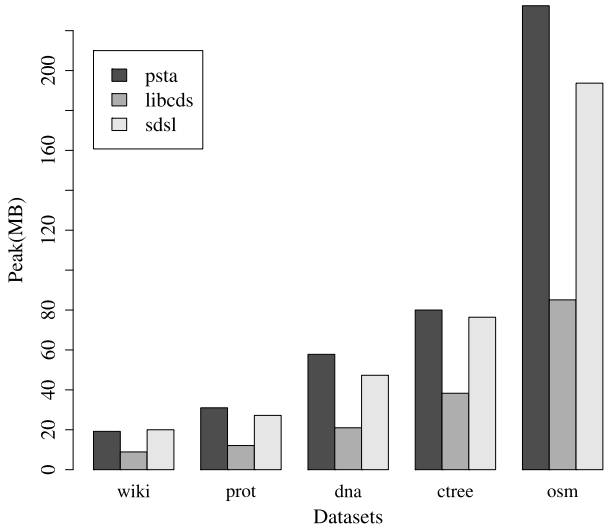


Fig. 9. Memory consumption of the algorithms `psta`, `libcds` and `sdsl`.

The differences in running times of the `psta` algorithm on one core and `seq` are insignificant. This implies that the overhead of the scheduler is negligible. The `psta` algorithm on a single core and `sdsl` outperformed `libcds` by an order of magnitude. One of the reasons for this is that `libcds` implements a different version of `RMMT` including *rank* and *select* structures, while `psta` and `sdsl` do not. Constructing these structures is costly. On a single core, `sdsl` was about 1.5 times faster than `psta`, but neither `sdsl` nor `libcds` were able to take advantage of multiple cores, so `psta` outperformed both of them starting at $p = 2$. The advantage of `sdsl` over `psta` on a single core, in spite of implementing essentially the same algorithm, can be attributed to the lack of tuning of `psta`.

Up to 16 cores, the speed-up of `psta` with `ctree` and `osm` datasets is almost linear whenever p is a power of 2 and the efficiency (speed-up/p) is 70% or higher with respect to `seq` and 60% with respect to `sdsl`, except for `ctree` on 32 cores. This is very good for a multicore architecture. When p is not a power of 2, speed-up is slightly worse. The reason is that, when p is a power of 2, `psta` can assign exactly one subtree to each thread (see Algorithm 4), distributing the work homogeneously across cores without any work stealing. When the number of threads is not a power of two, some threads have to process more than one subtree and other threads process only one, which degrades performance due to the overhead of work stealing.

There were three other factors that limited the performance of `psta` in our experiments: input size and resource contention with the OS.

Input size. For the two largest inputs we tested, `osm` and `ctree`, speed-up kept increasing as we added more cores. For `wiki`, `prot` and `dna`, however, the best speed-up were achieved with 28, 36 and 44 cores, respectively. Beyond this, the amount of work to be done per thread was small enough that the scheduling overhead caused by additional threads started to outweigh the benefit of reducing the processing time per thread further.

Resource contention. For $p < 64$, at least one core on our machine was available to OS processes, which allowed the remaining cores to be used exclusively by `psta`. For $p = 64$, `psta` competed with the OS for available cores. This had a detrimental effect on the efficiency of `psta` for $p = 64$.

The network topology of our machine *may* also impact in the performance of our algorithm. The four processors on our machine were connected in a grid topology [41]. Up to 32 threads, all threads can be run on a single processor or on two adjacent processors in the grid, which keeps the cost of communication between threads low. Beyond 32 threads, at least three processors are needed and at least two of them are not adjacent in the grid. This may increase the cost of communication between threads on these processors. In Section 4.5, we will discuss about the relationship of the topology of multicore machines and the performance of parallel algorithms.

4.3. Memory usage

We measured the amount of working memory (i.e., memory not occupied by the raw parenthesis sequence) used by `psta`, `libcds`, and `sdsl`. We did this by monitoring how much memory was allocated/released with `malloc`/`free` and recording the peak usage. For `psta`, we only measured the memory usage for $p = 1$. The extra memory needed for thread scheduling when $p > 1$ was negligible (less than 2% of extra memory for large p and large datasets). The results are shown in the Fig. 9. Even though `psta` uses more memory than both `libcds` and `sdsl`, the difference between `psta` and `sdsl` is a factor of less than 1.3. The difference between `psta` and `libcds` is no more than a factor of three and is outweighed

Table 4
Running times of PFEA algorithm, in seconds. seq corresponds to the sequential execution of PFEA. Columns with the superscript +16 and +32 represent the running times of PFEA algorithm by artificially increasing the workload with 16 and 32 CAS operations per edge, respectively. The best parallel times are shown using bold typeface.

p	ctree25	prot	dna	ctree25^{+16}	prot^{+16}	dna^{+16}	ctree25^{+32}	prot^{+32}	dna^{+32}
seq	5.67	52.87	31.33	55.04	473.92	275.06	102.94	887.61	514.73
1	6.07	34.83	57.29	54.99	275.23	474.07	102.80	514.09	886.01
4	1.68	9.35	15.62	14.22	70.84	121.92	26.15	130.30	224.86
8	1.05	5.77	9.80	7.33	36.39	62.79	13.32	66.35	114.42
12	0.87	3.99	6.90	5.02	25.18	43.23	8.99	45.02	77.49
16	0.73	3.66	5.57	3.91	19.59	33.50	6.86	34.37	59.15
20	0.76	3.42	5.53	3.21	16.06	27.61	5.58	28.16	48.26
24	0.76	3.41	5.25	2.77	13.91	24.03	4.73	23.96	41.04
28	0.66	3.28	5.36	2.47	12.40	21.23	4.16	20.87	36.05
32	0.67	3.37	5.71	2.30	12.52	19.49	4.08	18.62	35.22
36	0.68	3.23	5.57	2.27	11.61	19.95	3.71	18.70	32.29
40	0.68	3.15	5.48	2.16	10.91	18.75	3.38	17.17	29.59
44	0.68	3.16	5.44	2.04	10.20	17.70	3.19	15.98	27.56
48	0.69	**2.97**	5.19	1.92	9.67	16.79	3.00	15.02	25.82
52	0.64	3.05	5.37	1.84	9.23	16.01	2.83	14.12	24.33
56	0.65	3.01	5.33	1.79	8.83	15.18	2.71	13.37	22.87
60	**0.61**	3.19	5.35	**1.72**	8.46	**14.75**	2.58	12.63	21.82
64	0.71	3.05	**5.18**	1.80	**8.29**	15.13	**2.55**	**12.32**	**20.90**

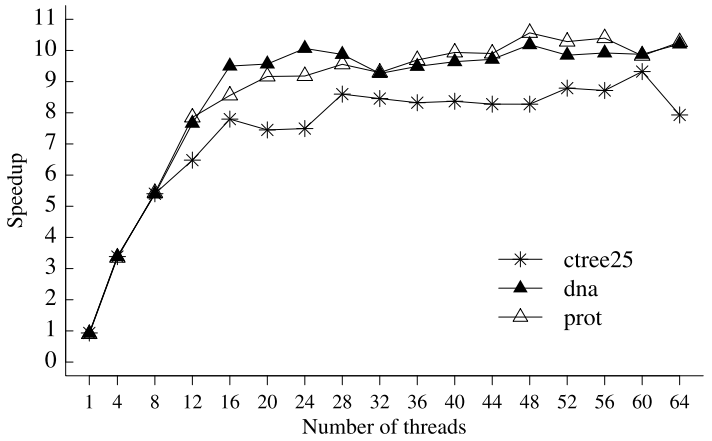


Fig. 10. Speedup of the PFEA algorithm with datasets ctree25, dna and prot.

by the substantially worse performance of libcds. The reduced working space used by libcds is due to the fact that its implementation does not store the array of excess values. Instead, libcds stores rank/select structures over the input bit vector P, computing excess values with $excess(i) := 2 \times rank_1(P, i) - i$, where $rank_1(P, i)$ gives the number of 1s on P up to the index i. Part of the higher memory usage of psta stems from the allocation of e', m' and M' arrays which store the partial excess values in the algorithm. Storing these values, however, is a key factor that helps psta achieves very good performance. The space used by our algorithm can be reduced by storing local excess values in the array e', instead of global values. However, reducing the space in such way will complicate the implementation of the queries over the RMMT.

4.4. Experimental results of the PFEA algorithm

Table 4 shows the running times of the PFEA algorithm with the datasets prot, dna and an additional dataset ctree25 that corresponds to a complete binary tree of depth 25. To compute the speedups, we used times obtained by seq. The best parallel times are identified using a bold typeface.

Fig. 10 shows the corresponding speedup of the PFEA algorithm. Up to 16 threads, the speedup is almost linear, obtaining at least 49% of efficiency (speedup/p) for the ctree25 dataset, that is, our algorithm reaches at least 49% of the linear speedup (the ideal). With more than 16 threads, the performance of our algorithm is poor, reaching at most 16% of efficiency for the prot algorithm and 64 threads. The poor efficiency of our algorithm is not explained by the DYM model. We think that it can be explained by its low workload. Algorithms with a low workload do not scale properly since the workload of their parallel tasks is not enough to pay the overhead of thread scheduling and memory transfers. In the case of the PFEA algorithm, each edge takes part of only a few comparisons and assignments. Therefore, the workload for each parallel task is not enough to take advantage of the 64 threads, even when we create $\Theta(p)$ parallel tasks. To demonstrate that

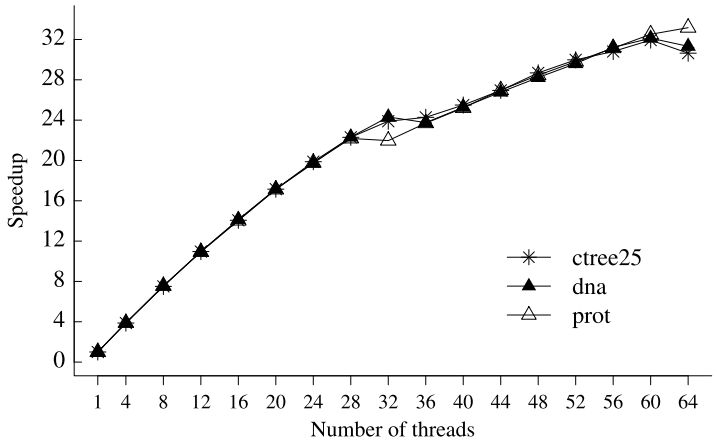


Fig. 11. Speedup of the PFEA algorithm with datasets ctree25, dna and prot, artificially increasing the workload with 16 CAS operations per edge.

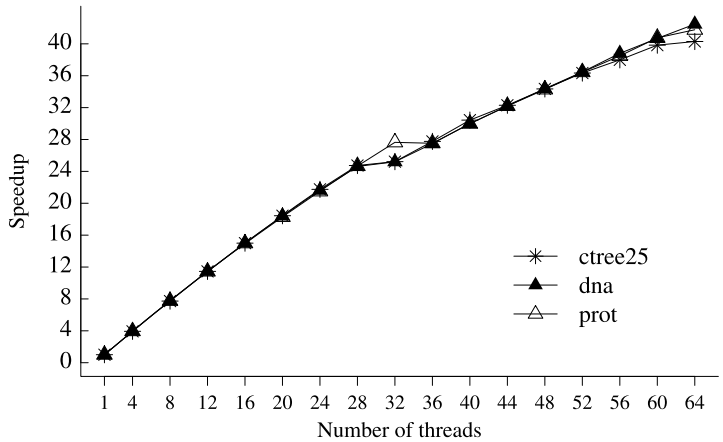


Fig. 12. Speedup of the PFEA algorithm with datasets ctree25, dna and prot, artificially increasing the workload with 32 CAS operations per edge.

the low workload is the reason of the low efficiency, we increased artificially the workload of our implementation. Between the lines 5 and 9, we added 16 and 32 CAS operations. On each iteration of the loop of line 5, each CAS operation was executed over $ET[i]$, increasing the workload for each edge. The complexity and correctness of our algorithm do not change with the addition of these extra operations. Columns 5–10 of Table 4 show the resulting running times after adding 16 and 32 extra operations. Figs. 11 and 12 show the corresponding speedup. With 16 extra operations, the efficiency was at least 48% for the ctree25 dataset and 64 threads. For 16 threads, the efficiency increased, reaching a 88% for the ctree25 dataset. With 32 extra operations, the efficiency was at least 63% up to 64 threads and 94% up to 16 threads.

Another factor that, we think, limited the performance of the PFEA algorithm was the topology of the experiment. As was mentioned before, our machine has four processors connected in a grid topology, which involves communication costs among processors. Each processor executes up to 16 threads. We observe that in the Fig. 10, the algorithm scales up to 16 threads. With more threads, the communication costs may affect the scalability. With more workload, Fig. 11 shows a linear scalability up to 32 threads. After 32 threads, the efficiency of the algorithm decreases. For the experiment with 32 extra operations, Fig. 12 shows a similar behaviour, with the difference that after 32 threads, the efficiency is better than in Fig. 11. For 64 threads, all the speedups have a slowdown, since the PFEA algorithm has to compete with the OS for the available cores. In Section 4.5 we will discuss more about the effects of the machine topology in the performance of the PFEA algorithm.

4.5. Discussion

For domains where trees have billions of nodes, the psta algorithm exhibits a good speed-up up to 64 cores. The speed-up is degraded for trees with fewer nodes. However, even in such cases, our algorithm reaches good speed-up up to 32 cores. Additionally, our algorithm outperforms the state-of-the-art implementations using only $p = 2$ threads. Considering all of this, the psta algorithm is a good option to construction succinct trees in commodity multicore architectures.

With respect to the working space, our `psta` algorithm is competitive with `sdsl` and it does not use more than three times the memory used by `libcds`, which is the slowest algorithm. Despite our algorithm using more memory than `sdsl` and `libcds`, it is up to 20 times and 376 times faster with 64 cores, respectively.

The scalability of the `PFEA` algorithm up to 16 threads is good in practice (nearly 50% efficiency). However, the lack of workload of our algorithm prevents it from obtaining a good practical scalability with more threads. In this kind of algorithms, we cannot expect a better scalability adding more threads. Nevertheless, this poses an interesting problem: For a given algorithm, find the maximum number of threads that achieve at least a 50% efficiency. Once we find such number, the rest of the threads may be used potentially in other procedures. The implementation of multicore algorithms is non-trivial, since it needs to take care about the communication costs, memory hierarchy, cache coherency, etc, which may affect the performance. Therefore, we consider a parallel algorithm with a 50% efficiency a good parallel implementation.

It is important to emphasize that this work involves what we think are important contributions to practical implementations of parallel algorithms in commodity architectures. In a way, then, topology has become important again, though thankfully not a show-stopper. For example, in the Figs. 7, 8 and 10, we observed that algorithms had a slowdown of the speedup at 16, 32, 48 and 64 threads. We hypothesize that the factor that generated the slowdown of the speedups of the algorithms had to do with the topology of the machine where we ran our experiments. The four processors on our machine were connected in a grid topology [41]. This increases the cost of communication between threads on these processors. Additionally, there exist other factors of the architecture that can impact the performance of multicore algorithms to construct succinct data structures, such as, cache inclusion policy which may vary for each new architecture, special wiring among cores and among caches, and cache coherency protocol. The impact of all of these factors in the implementation of multicore construction algorithms need to be studied in more detail.

5. Conclusions and future work

In this paper, we demonstrated that it is possible to improve the construction time of succinct trees using multicore parallelism. We introduced a practical algorithm with $O(n)$ work and $O(\lg n)$ span, to construct a succinct representation of a tree with n nodes. This representation supports a rich set of operations in $O(\lg n)$ time. Our algorithm substantially outperformed state-of-the-art sequential constructions of this data structure, achieved very good speed-up up to 64 cores, and is to the best of our knowledge the first parallel construction algorithm of a succinct representation of ordinal trees. We also presented a second version which supports queries in $O(c)$ time, with $O(n+\frac{n}{\lg^c n}\lg(\frac{n}{\lg^c n})+c^c)$ work and $O(c+\lg(\frac{nc^c}{\lg^c n}))$ span.

While we focused on representing static trees succinctly in this paper, the approach we have taken may also extend to the construction of *dynamic* succinct trees (e.g., [2]), of succinct representations of *labelled* trees, and of other succinct data structures that use succinct trees as building blocks (e.g., the succinct representation of planar graphs).

References

[1] L. Ferres, J. Fuentes-Sepúlveda, M. He, N. Zeh, Parallel construction of succinct trees, in: E. Bampis (Ed.), Experimental Algorithms, in: Lecture Notes in Computer Science, vol. 9125, Springer International Publishing, 2015, pp. 3–14.
[2] G. Navarro, K. Sadakane, Fully functional static and dynamic succinct trees, ACM Trans. Algorithms 10 (3) (2014) 16:1–16:39.
[3] G. Jacobson, Space-efficient static trees and graphs, in: FOCS, 1989, pp. 549–554.
[4] D.R. Clark, J.I. Munro, Efficient suffix trees on secondary storage, in: SODA, 1996, pp. 383–391.
[5] M. He, J.I. Munro, S.R. Satti, Succinct ordinal trees based on tree covering, ACM Trans. Algorithms 8 (4) (2012) 42.
[6] A. Farzan, J.I. Munro, A uniform paradigm to succinctly encode various families of trees, Algorithmica 68 (1) (2014) 16–40.
[7] R. Raman, S.S. Rao, Succinct representations of ordinal trees, in: Space-Efficient Data Structures, Streams, and Algorithms, 2013, pp. 319–332.
[8] G. Navarro, Compact Data Structures: A Practical Approach, Cambridge University Press, 2016.
[9] J. Fuentes-Sepúlveda, E. Elejalde, L. Ferres, D. Seco, Parallel construction of wavelet trees on multicore architectures, Knowl. Inf. Syst. (2017) 1043–1066, http://dx.doi.org/10.1007/s10115-016-1000-6.
[10] J. Shun, Parallel wavelet tree construction, in: Proceedings of the 2015 Data Compression Conference, DCC '15, IEEE Computer Society, Washington, DC, USA, 2015, pp. 63–72.
[11] J. Labeit, J. Shun, G.E. Blelloch, Parallel lightweight wavelet tree, suffix array and fm-index construction, in: 2016 Data Compression Conference, DCC 2016, Snowbird, UT, USA, March 30–April 1, 2016, 2016, pp. 33–42.
[12] J. Labeit, J. Shun, G.E. Blelloch, Parallel lightweight wavelet tree, suffix array and fm-index construction, J. Discrete Algorithms 43 (2017) 2–17, http://dx.doi.org/10.1016/j.jda.2017.04.001.
[13] D. Arroyuelo, R. Cánovas, G. Navarro, K. Sadakane, Succinct trees in practice, in: ALENEX, SIAM, Austin, TX, USA, 2010, pp. 84–97.
[14] M.A. Bender, M. Farach-Colton, The level ancestor problem simplified, Theoret. Comput. Sci. 321 (1) (2004) 5–12, http://dx.doi.org/10.1016/j.tcs.2003.05.002.
[15] J.I. Munro, V. Raman, Succinct representation of balanced parentheses, static trees and planar graphs, in: FOCS, 1997, pp. 118–126.
[16] D. Benoit, E.D. Demaine, J.I. Munro, V. Raman, Representing trees of higher degree, in: WADS, in: Lecture Notes in Computer Science, vol. 1663, Springer-Verlag, 1999, pp. 169–180.
[17] R.F. Geary, R. Raman, V. Raman, Succinct ordinal trees with level-ancestor queries, in: SODA, 2004, pp. 1–10.
[18] J. Jansson, K. Sadakane, W.-K. Sung, Ultra-succinct representation of ordered trees with applications, in: Games in Verification, J. Comput. System Sci. 78 (2) (2012) 619–631.
[19] H.-I. Lu, C.-C. Yeh, Balanced parentheses strike back, ACM Trans. Algorithms 4 (2008) 28:1–28:13.
[20] K. Sadakane, Compressed suffix trees with full functionality, Theory Comput. Syst. 41 (4) (2007) 589–607.
[21] J. Munro, V. Raman, S. Rao, Space efficient suffix trees, J. Algorithms 39 (2) (2001) 205–222.
[22] J.I. Munro, R. Raman, V. Raman, S.S. Rao, Succinct representations of permutations and functions, Theoret. Comput. Sci. 438 (2012) 74–88, http://dx.doi.org/10.1016/j.tcs.2012.03.005.

[23] M. Patrascu, Succincter, in: FOCS, FOCS '08, IEEE Computer Society, Washington, DC, USA, 2008, pp. 305–313.
[24] M.L. Fredman, D.E. Willard, Surpassing the information theoretic bound with fusion trees, J. Comput. System Sci. 47 (3) (1993) 424–436.
[25] J. Fischer, V. Heun, A new succinct representation of RMQ-information and improvements in the enhanced suffix array, in: Proceedings of the First International Conference on Combinatorics, Algorithms, Probabilistic and Experimental Methodologies, ESCAPE'07, Springer-Verlag, Berlin, Heidelberg, 2007, pp. 459–470.
[26] K. Sadakane, Succinct representations of lcp information and improvements in the compressed suffix arrays, in: Proceedings of the Thirteenth Annual ACM–SIAM Symposium on Discrete Algorithms, SODA '02, SIAM, Philadelphia, PA, USA, 2002, pp. 225–232.
[27] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, Introduction to Algorithms, 3rd edition, The MIT Press, 2009, pp. 772–812, Ch. Multithreaded Algorithms.
[28] R.D. Blumofe, C.E. Leiserson, Scheduling multithreaded computations by work stealing, J. ACM 46 (5) (1999) 720–748.
[29] D.R. Helman, J. JáJá, Prefix computations on symmetric multiprocessors, J. Parallel Distrib. Comput. 61 (2) (2001) 265–278.
[30] J. Vuillemin, A unifying look at data structures, Commun. ACM 23 (4) (1980) 229–239, http://dx.doi.org/10.1145/358841.358852.
[31] J. Shun, G.E. Blelloch, A simple parallel cartesian tree algorithm and its application to parallel suffix tree construction, ACM Trans. Parallel Comput. 1 (1) (2014) 8:1–8:20.
[32] R. Grossi, A. Gupta, J. Vitter, High-order entropy-compressed text indexes, in: SODA, SIAM, Philadelphia, PA, USA, 2003, pp. 841–850.
[33] C. Makris, Wavelet trees: a survey, Comput. Sci. Inf. Syst. 9 (2) (2012) 585–625.
[34] G. Navarro, Wavelet trees for all, in: J. Kärkkäinen, J. Stoye (Eds.), Combinatorial Pattern Matching, in: Lecture Notes in Computer Science, vol. 7354, Springer, Berlin, Heidelberg, 2012, pp. 2–26.
[35] R. Raman, The power of collision: randomized parallel algorithms for chaining and integer sorting, in: Proceedings of the Tenth Conference on Foundations of Software Technology and Theoretical Computer Science, Springer-Verlag, London, UK, 1990, pp. 161–175.
[36] B.-F. Wang, G.-H. Chen, Cost-optimal parallel algorithms for constructing b-trees, Inform. Sci. 81 (1) (1994) 55–72.
[37] J. Fischer, V. Heun, Space-efficient preprocessing schemes for range minimum queries on static arrays, SIAM J. Comput. 40 (2) (2011) 465–492.
[38] F. Claude, A compressed data structure library, 2015, last accessed: January 17.
[39] S. Gog, Succinct data structure library 2.0, 2015, last accessed: January 17.
[40] T. Bingmann, malloc_count – tools for runtime memory usage analysis and profiling, 2015, last accessed: January 17.
[41] U. Drepper, What every programmer should know about memory, 2007.