Parallel Construction of Succinct Trees*

Leo Ferres[1], José Fuentes-Sepúlveda[1], Meng He[2], and Norbert Zeh[2]

[1] Department of Computer Science, Universidad de Concepción, Chile,
{lferres,jfuentess}@udec.cl
[2] Faculty of Computer Science, Dalhousie University, Canada,
{mhe,nzeh}@cs.dal.ca

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 a practical parallel algorithm that improves the state of the art in succinct tree construction. Given a tree on n nodes stored as a sequence of balanced parentheses, our algorithm builds a succinct tree representation in $O(n/p + \lg p)$ time, where p is the number of available cores. The constructed representation uses $2n + o(n)$ bits of space and supports a rich set of operations in $O(\lg n)$ time. In experiments using up to 64 cores and on inputs of different sizes, our algorithm achieved good parallel speed-up. We also present an algorithm that takes $O(n/p + \lg p)$ time to construct the balanced parenthesis representation 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, which is close to the information-theoretic lower bound and much less than the space used by traditional pointer-based representations.

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.

* This work was supported by the Emerging Leaders in the Americas scholarship programme, NSERC, and the Canada Research Chairs programme.

In [10], two practical multicore algorithms for wavelet tree construction were introduced. Both algorithms perform $O(n \lg \sigma)$[3] work and have span $O(n)$, where n is the input size and σ is the alphabet size. In [21], Shun introduced three new algorithms to construct wavelet trees in parallel. Among these three algorithms, the best algorithm in practice performs $O(n \lg \sigma)$ work and has span $O(\lg n \lg \sigma)$. Shun also explained how to parallelize the construction of rank/select structures so that it requires $O(n)$ work and $O(1)$ 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 [19] in $O(n/p + \lg p)$ time using p cores. This structure 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 [1]. 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.

2 Preliminaries

Succinct trees. Jacobson [15] 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 [5] 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 [2, 9, 11, 13, 16–19]. See [20] for a thorough survey.

Navarro and Sadakane [19] recently 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 Appendix A); only the work in [9, 13] supports two additional operations. An experimental study of succinct trees [1] 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.

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

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 $2n$.

The NS-representation is not the first structure to use balanced parentheses to represent trees. Munro and Raman [18] 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 supported by the NS-representation. Additional operations can be supported using auxiliary data structures [17], but supporting all operations in Appendix A 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, these primitive operations are the following, 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:

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

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)[4] 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.

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

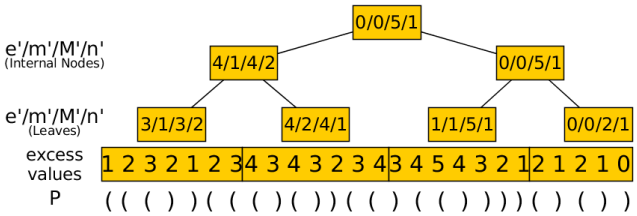

Fig. 1: Range min-max tree

Next we briefly review the `RMMT` structure and how it supports the primitive operations for $g = \pi$. Navarro and Sadakane [19] 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 version 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 Figure 1). (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 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 containing $P[i]$, we report it. Otherwise, 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 the answer. If so, we 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 the answer 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 $O(\lg n)$.

Supporting operations on the leaves, such as finding the ith leaf from the left, reduces to `rank` and `select` operations over a bit vector $P_1[1..2n]$ where $P_1[i] = 1$ iff $P[i] = 1$ and $P[i+1] = 0$. `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.

To analyze the space usage, 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 `RMMT` is thus $2n + O(n/\lg n)$ bits.

Dynamic multithreading (DyM) model. In the DyM model [7, 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 [4]. 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 overhead is logarithmic in the number of iterations. When a procedure exits, it implicitly performs a **sync** to ensure all threads it spawned finish first.

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, and known as the "folklore encoding". For trees whose folklore encoding is not directly available, Appendix B describes 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 [4], 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.

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 1), 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 2), the algorithm computes the entries of m', M' and n' corresponding to internal nodes of the RMMT. In the third phase (Algorithm 3), 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 1), 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 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 [14] in line 22 to compute the global excess values

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 1: 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 to 0 do
8      parfor d := 0 to 2^l − 1 do
9          i := d + 2^l − 1
10         concat(i, m', M', n')
```

Algorithm 2: PSTA (part II)

```
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 3: 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

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 2) 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 with root 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 3 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). 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.

Theoretical analysis. Lines 1–21 of Algorithm 1 require $O(n)$ work and have span $O(n/p)$. Line 22 requires $O(p)$ work and has span $O(\lg p)$ because we compute a prefix sum over only p values. Lines 23–28 require $O(n/s)$ work and have span $O(n/sp)$. Lines 1–6 of Algorithm 2 require $O(n/s)$ work and have span $O(n/sp)$. Lines 7–10 require $O(p)$ work and have span $O(\lg p)$. Algorithm 3 requires $O(\sqrt{2^w}\texttt{poly}(w))$ work and has span $O(\sqrt{2^w}\texttt{poly}(w)/p)$. As was defined in Section 2, w is the machine word size. Thus, the total work of `PSTA` is $T_1 = O(n + \lg p + \sqrt{2^w}\texttt{poly}(w))$ and its span is $O(n/p + \lg p + \sqrt{2^w}\texttt{poly}(w)/p)$. For $p \to \infty$, we get a span of $T_\infty = O(\lg n)$. This gives a running time of $T_p = O(T_1/p + T_\infty) = O(n/p + \lg p + \sqrt{2^w}\texttt{poly}(w)/p)$ on p cores. The speedup is $T_1/T_p = O\left(\frac{p(n+\sqrt{2^w}\texttt{poly}(w))}{n+\sqrt{2^w}\texttt{poly}(w)+p\lg p}\right)$. 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 $\frac{n+\sqrt{2^w}\texttt{poly}(w)}{\lg n}$.

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 [4]. 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. *A $(2n + o(n))$-bit representation of an ordinal tree on n nodes and its balanced parenthesis sequence can be computed in $O(n/p + \lg p)$ time using $O(n \lg n)$ bits of working space, where p is the number of cores. This representation can support the operations in Appendix A in $O(\lg n)$ time.*

4 Experimental Results

To evaluate the performance of our `PSTA` algorithm, we compare it against `libcds` [6] and `sdsl` [12], 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 in 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. We implemented the `PSTA` algorithm in C and compiled it using GCC 4.9 with optimization level -O2 and using the -ffast-math flag.[5] All parallel code was compiled using the GCC Cilk branch. The same flags were used to compile `libcds` and `sdsl`, which were written in C++.

We tested our algorithm on five inputs. The first two were suffix trees of the DNA (`dna`, 1,154,482,174 parentheses), and protein (`prot`, 670,721,006 parentheses) data from the Pizza & Chili corpus[6]. These suffix trees were constructed using code from `http://www.daimi.au.dk/~mailund/suffix_tree.html`. The next two inputs were XML trees of the Wikipedia dump[7] (`wiki`, 498,753,914 parentheses) and OpenStreetMap dump[8] (`osm`, 4,675,776,358 parentheses). The final input was a complete binary tree of depth 30 (`ctree`, 2,147,483,644 parentheses).

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

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` [3]. In our experiments, the chunk size s was fixed at 256.

[5] The code and data needed to replicate our results are available at `http://www.inf.udec.cl/~josefuentes/sea2015`.

[6] `http://pizzachili.dcc.uchile.cl`

[7] `http://dumps.wikimedia.org/enwiki/20150112/enwiki-20150112-pages-articles.xml.bz2` (January 12, 2015)

[8] `http://wiki.openstreetmap.org/wiki/Planet.osm` (January 10, 2015)

p	wiki	prot	dna	ctree	osm
libcds	33.16	44.24	75.87	140.41	339.21
sdsl	1.93	2.66	4.57	8.35	18.10
1	2.89	4.22	7.21	12.16	30.60
2	1.44	2.13	3.64	6.15	15.43
4	.73	1.10	1.87	3.18	7.98
8	.37	.57	.98	1.59	4.14
16	.25	.35	.58	.86	2.21
32	.18	.25	.39	.63	1.33
64	.27	.29	.39	.48	1.01

Table 1: Running times of PSTA, libcds, and sdsl in seconds.

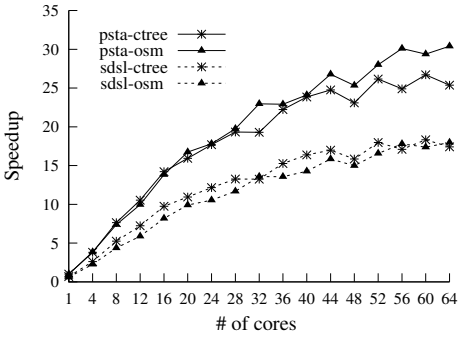

Fig. 2: Speed-up on ctree and osm data sets.

Running time and speed-up. Table 1 shows the wall clock times achieved by psta, libcds, and sdsl on different inputs. Each time is the minimum achieved over three 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. Figure 2 shows the speed-up for the ctree and osm inputs compared to the running times of psta on a single core and of sdsl.

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 (1) lack of tuning of psta and (2) some overhead with running parallel code on a single core.

Up to 16 cores, the speed-up of psta is almost linear whenever p is a power of 2 and the efficiency (speed-up/p) is 70% or higher, 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 2), 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: network topology, input size, and resource contention with the OS.

Topology. The four processors on our machine were connected in a grid topology [8]. 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 processor are needed

and at least two of them are not adjacent in the grid. This increases the cost of communication between threads on these processors noticeably.

Input size. For the two largest inputs we tested, `osm` and `ctree`, speed-up kept increasing as we added more cores. For `wiki`, however, the best speed-up was achieved with 36 cores. 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 $p = 64$.

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. Due to lack of space, we report the results only for the two largest inputs, `ctree` and `osm`. For the `ctree` input, `psta`, `libcds`, and `sdsl` used 112MB, 38MB, and 76MB of memory, respectively. For `osm`, they used 331MB, 85MB, and 194MB, respectively. Even though `psta` uses more memory than both `libcds` and `sdsl`, the difference between `psta` and `sdsl` is a factor of less than two. The difference between `psta` and `libcds` is no more than a factor of four and is outweighed by the substantially worse performance of `libcds`.

Part of the higher memory usage of `psta` stems from the allocation of e', m', M' and n' arrays to store the partial excess values in the algorithm. Storing these values, however, is a key factor that helps `psta` to achieve very good performance.

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 that takes $O(n/p + \lg p)$ time to construct a succinct representation of a tree with n nodes using p threads. 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.

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., [19]), 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).

Acknowledgements. We would like to thank Diego Arroyuelo, Roberto Asín, and Rodrigo Cánovas for their time and making resources available to us.

References

1. Arroyuelo, D., Cánovas, R., Navarro, G., Sadakane, K.: Succinct trees in practice. In: ALENEX. pp. 84–97. SIAM Press, Austin, Texas, USA (2010)
2. Benoit, D., Demaine, E.D., Munro, J.I., Raman, V.: Representing trees of higher degree. In: WADS. pp. 169–180. Springer-Verlag LNCS 1663 (1999)
3. Bingmann, T.: malloc_count - tools for runtime memory usage analysis and profiling, last accessed: January 17, 2015
4. Blumofe, R.D., Leiserson, C.E.: Scheduling multithreaded computations by work stealing. J. ACM 46(5), 720–748 (1999)
5. Clark, D.R., Munro, J.I.: Efficient suffix trees on secondary storage. In: SODA. pp. 383–391 (1996)
6. Claude, F.: A compressed data structure library, last accessed: January 17, 2015
7. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, chap. Multithreaded Algorithms, pp. 772–812. The MIT Press, third edn. (2009)
8. Drepper, U.: What every programmer should know about memory (2007)
9. Farzan, A., Munro, J.I.: A uniform paradigm to succinctly encode various families of trees. Algorithmica 68(1), 16–40 (2014)
10. Fuentes-Sepúlveda, J., Elejalde, E., Ferres, L., Seco, D.: Efficient wavelet tree construction and querying for multicore architectures. In: SEA. pp. 150–161. Springer International Publishing (2014)
11. Geary, R.F., Raman, R., Raman, V.: Succinct ordinal trees with level-ancestor queries. In: SODA. pp. 1–10 (2004)
12. Gog, S.: Succinct data structure library 2.0, last accessed: January 17, 2015
13. He, M., Munro, J.I., Satti, S.R.: Succinct ordinal trees based on tree covering. ACM Trans. Algorithms 8(4), 42 (2012)
14. Helman, D.R., JáJá, J.: Prefix computations on symmetric multiprocessors. J. Par. Dist. Comput. 61(2), 265 – 278 (2001)
15. Jacobson, G.: Space-efficient static trees and graphs. In: FOCS. pp. 549–554 (1989)
16. Jansson, J., Sadakane, K., Sung, W.K.: Ultra-succinct representation of ordered trees. In: SODA (2007)
17. Lu, H.I., Yeh, C.C.: Balanced parentheses strike back. ACM Trans. Algorithms 4, 28:1–28:13 (July 2008)
18. Munro, J.I., Raman, V.: Succinct representation of balanced parentheses, static trees and planar graphs. In: FOCS. pp. 118–126 (1997)
19. Navarro, G., Sadakane, K.: Fully functional static and dynamic succinct trees. ACM Trans. Algorithms 10(3), 16:1–16:39 (May 2014)
20. Raman, R., Rao, S.S.: Succinct representations of ordinal trees. In: Space-Efficient Data Structures, Streams, and Algorithms. pp. 319–332 (2013)
21. Shun, J.: Parallel wavelet tree construction. CoRR abs/1407.8142 (2014)

Appendices

A Operations supported by the NS-representation

Operation	Description
`child`(x, i)	Find the ith child of node x
`child_rank`(x)	Report the number of left siblings of node x
`degree`(x)	Report the degree of node x
`depth`(x)	Report the depth of node x
`level_anc`(x, i)	Find the ancestor of node x that is i levels above node x
`subtree_size`(x)	Report the number of nodes in the subtree rooted at node x
`height`(x)	Report the height of the subtree rooted at x
`deepest_node`(x)	Find the deepest node in the subtree rooted at node x
`LCA`(x, y)	Find the lowest common ancestor of nodes x and y
`lmost_leaf`(x) /`rmost_leaf`(x)	Find the leftmost/rightmost leaf of the subtree rooted at node x
`leaf_rank`(x)	Report the number of leaves before node x in preorder
`leaf_select`(i)	Find the ith leaf from left to right
`pre_rank`(x)/`post_select`(x)	Report the number of nodes preceding node x in preorder/postorder
`pre_select`/`post_select`(i)	Find the ith node in preorder/postorder
`level_lmost`(i)/`level_rmost`(i)	Find the leftmost/rightmost node among all nodes at depth i
`level_succ`(x)/`level_pred`(x)	Find the node immediately to the left/right of node x among all nodes at depth i
`access`(i)	Report $P[i]$
`find_open`(i)/`find_close`(i)	Find The matching parenthesis of $P[i]$
`enclose`(i)	Find the closest enclosing matching parenthesis pair for $P[i]$
`rank_open`(i)/`rank_close`(i)	Report the number of opening/closing parentheses in $P[1..i]$
`select_open`(i)/`select_close`(i)	Find the ith opening/closing parenthesis

Table 2: Operations supported by the NS-representation [19], including operations over the corresponding balanced parenthesis sequence.

B Parallel Folklore Encoding Algorithm

The PSTA algorithm requires the input tree T to be given in the form of a balanced parenthesis sequence P, but in many 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 $next(e.x)$ and $first(e.x)$ to denote the indices in E of e's successor and of the first element in x's adjacency list, respectively. Both are easily computed in constant time by following pointers.

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 4: PFEA

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, then 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 4 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 (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 [14]. 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.

Lines 4–16 and 18–22 perform $O(n)$ work and have span $O(n/p)$. 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. Line 17 performs $O(n)$ work and has span $O(\lg p + n/p)$. 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 $T_p = O(n/p + \lg p)$.