Parallel Construction of Succinct Trees

José Fuentes Sepúlveda
Department of Computer Science
University of Concepción. Concepción, Chile
Email: jfuentess@udec.cl

***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.**

***Index Terms*—Succinct Tree, Multicore, Parallel Algorithms**

I. Introduction

Trees are ubiquitous in Computer Science. 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.

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 [1], [2].

In this paper, we provide a parallel algorithm that constructs the `RMMT` tree representation of [3] 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. 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 and achieves good speed-up.

II. Preliminaries

Jacobson [4] 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 basic operations 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 $\Theta(\lg n)$. See [6] for a thorough survey.

Navarro and Sadakane [3] 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 NS-representation supports a large number of constant time navigational operations (see Table 1 of [3]). An experimental study of succinct trees [7] 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.

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 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* (See [3] for more details). To do so, Navarro and Sadakane designed a simple data structure called *Range Min-Max Tree* (`RMMT`), which supports the primitive operations 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.

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. The `RMMT` is a complete binary tree over the sequence of chunks. 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. The representation of the `RMMT` consists of four arrays e', m', M', and n' of size $O(2n/s)$. The uth entry of each of these arrays stores the excess at the last position in P_u ($e'[u]$), the minimum and maximum excess at any position in P_u ($m'[u]$ and $M'[u]$) and the number of positions in P_u that have the minimum excess value $m'[u]$ ($n'[u]$).Combined with a standard technique called *table lookup*, an `RMMT` supports the primitive operations in $O(\lg n)$ time. The space needed by the `RMMT` is $2n + O(n/\lg n)$ bits.

III. Our Parallel Algorithm

In this section, we describe our new parallel algorithm called the *Parallel Succinct Tree Algorithm* (`PSTA`). Its input is the balanced parenthesis sequence P of an n-node tree T, known as the "folklore encoding". 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 [8], 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.

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 algorithmm (Algorithm 1) consists of three phases. In the first phase (lines 1–27), 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 (lines 28–35), the algorithm computes the entries of m', M' and n' corresponding to internal nodes of the RMMT. In the third phase (lines 36–47), 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, 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. 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 [9] 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 23–27).

The computation of entries of m', M', and n' 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 28–32). 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 33–35 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.

Lines 36–47 illustrate 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 primitive operation `fwd_search`. 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.

The theoretical analysis is done under the *Dynamic Multithreading Model (DyM)* [10]. Lines 1–21 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 28–32 require $O(n/s)$ work and have span $O(n/sp)$. Lines 33–35 require $O(p)$

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; up := low + s
9          for par := low to up − 1 do
10             e'_t += 2 * P[par] − 1
11             if e'_t < m'_t then
12                 m'_t := e'_t; n'_t := 1
13             else if e'_t = m'_t then
14                 n'_t += 1
15             else if e'_t > M'_t then
16                 M'_t := e'_t
17         e'[t * ct + chk] := e'_t
18         m'[t * ct + chk + o] := m'_t
19         M'[t * ct + chk + o] := M'_t
20         n'[t * ct + chk + o] := n'_t
21 parallel_prefix_sum(e', ct)
22 parfor t := 1 to threads − 1 do
23     for chk := 0 to ct − 1 do
24         if chk < ct − 1 then
25             e'[t * ct + chk] += e'[t * ct − 1]
26         m'[t * ct + chk + o] += e'[t * ct − 1]
27         M'[t * ct + chk + o] += e'[t * ct − 1]
28 parfor st := 0 to 2^⌈lg threads⌉ − 1 do
29     for l := ⌈lg(2n/s)⌉ − 1 downto ⌈lg threads⌉ do
30         for d := 0 to 2^(l−⌈lg threads⌉) − 1 do
31             i := d + 2^l − 1 + st * 2^(l−⌈lg threads⌉)
32             concat(i, m', M', n')
33 for l := ⌈lg threads⌉ − 1 to 0 do
34     parfor d := 0 to 2^l − 1 do
35         concat(d + 2^l − 1, m', M', n')
36 parfor x := −w to w − 1 do
37     parfor y := 0 to √(2^w) − 1 do
38         i := ((x + w) << w) OR w
39         near_fwd_pos[i] := w
40         p, excess := 0
41         repeat
42             excess += 1 − 2 * ((y AND(1 << p)) = 0)
43             if excess = x then
44                 near_fwd_pos[i] := p
45                 break
46             p += 1
47         until p ≥ w
```

Algorithm 1: PSTA

work and have span $O(\lg p)$. Third phase requires $O(\sqrt{2^w}\texttt{poly}(w))$ work and has span $O(\sqrt{2^w}\texttt{poly}(w)/p)$, with w 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 $O(p)$.

The PSTA algorithm 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 which is optimal within a constant factor

[8]. 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 $O(n \lg n)$. 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 Table 1 of [3] in $O(\lg n)$ time.

IV. Experimental Results

To evaluate the performance of our `PSTA` algorithm, we compare it against `libcds` [11] 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 III in that the prefix sum computation in line 21 of the algorithm is done sequentially in our 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. All parallel code was compiled using the GCC Cilk branch. (See [13] for to replicate our results).

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. 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 (`wiki`, 498,753,914 parentheses) and OpenStreetMap dump (`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>`, taking the minimum achieved over three non-consecutive runs. Memory usage was measured using the tools provided by `malloc_count` [14]. In our experiments, the chunk size s was fixed at 256.

Figure 1 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, which 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 $p = 2$.

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 (lines 28–35 of Algorithm 1), distributing the work homogeneously across cores without any work stealing.

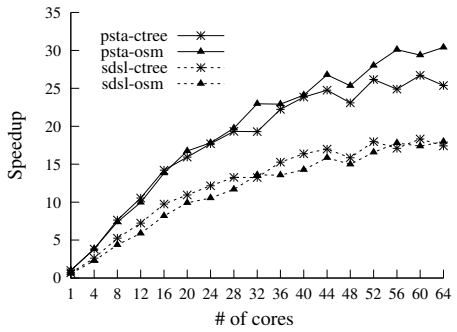

Fig. 1. Speed-up on `ctree` and `osm` data sets.

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 the performance.

The four processors on our machine were connected in a grid topology [15]. 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.

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.

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, which had a detrimental effect on the efficiency of `psta`.

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. 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 worse performance of `libcds`.

V. 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 ordinal trees.

While we focused on representing static trees succinctly, the approach we have taken may also extend to the construction of *dynamic* succinct trees, 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] J. Fuentes-Sepúlveda, E. Elejalde, L. Ferres, and D. Seco, "Efficient wavelet tree construction and querying for multicore architectures," in *SEA*. Springer International Publishing, 2014, pp. 150–161.

[2] J. Shun, "Parallel wavelet tree construction," *CoRR*, vol. abs/1407.8142, 2014.

[3] G. Navarro and K. Sadakane, "Fully functional static and dynamic succinct trees," *ACM Trans. Algorithms*, vol. 10, no. 3, pp. 16:1–16:39, May 2014.

[4] G. Jacobson, "Space-efficient static trees and graphs," in *FOCS*, 1989, pp. 549–554.

[5] D. R. Clark and J. I. Munro, "Efficient suffix trees on secondary storage," in *SODA*, 1996, pp. 383–391.

[6] R. Raman and S. S. Rao, "Succinct representations of ordinal trees," in *Space-Efficient Data Structures, Streams, and Algorithms*, 2013, pp. 319–332.

[7] D. Arroyuelo, R. Cánovas, G. Navarro, and K. Sadakane, "Succinct trees in practice," in *ALENEX*. Austin, Texas, USA: SIAM Press, 2010, pp. 84–97.

[8] R. D. Blumofe and C. E. Leiserson, "Scheduling multithreaded computations by work stealing," *J. ACM*, vol. 46, no. 5, pp. 720–748, 1999.

[9] D. R. Helman and J. JáJá, "Prefix computations on symmetric multiprocessors," *J. Par. Dist. Comput.*, vol. 61, no. 2, pp. 265 – 278, 2001.

[10] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd ed. The MIT Press, 2009, ch. Multithreaded Algorithms, pp. 772–812.

[11] F. Claude, "A compressed data structure library," last accessed: January 17, 2015.

[12] S. Gog, "Succinct data structure library 2.0," last accessed: January 17, 2015.

[13] L. Ferres, J. Fuentes-Seplveda, M. He, and N. Zeh, "Parallel construction of succinct trees," in *Experimental Algorithms*, ser. Lecture Notes in Computer Science, E. Bampis, Ed. Springer International Publishing, 2015, vol. 9125, pp. 3–14. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-20086-6_1

[14] T. Bingmann, "malloc_count - tools for runtime memory usage analysis and profiling," last accessed: January 17, 2015.

[15] U. Drepper, "What every programmer should know about memory," 2007.