REGULAR PAPER

Parallel construction of wavelet trees on multicore architectures

José Fuentes-Sepúlveda[1] · Erick Elejalde[1] · Leo Ferres[2] · Diego Seco[1]

Received: 16 September 2015 / Revised: 20 July 2016 / Accepted: 26 September 2016 /
Published online: 5 October 2016

Abstract The wavelet tree has become a very useful data structure to efficiently represent and query large volumes of data in many different domains, from bioinformatics to geographic information systems. One problem with wavelet trees is their construction time. In this paper, we introduce two algorithms that reduce the time complexity of a wavelet tree's construction by taking advantage of nowadays ubiquitous multicore machines. Our first algorithm constructs all the levels of the wavelet in parallel with $O(n)$ time and $O(n \lg \sigma + \sigma \lg n)$ bits of working space, where n is the size of the input sequence and σ is the size of the alphabet. Our second algorithm constructs the wavelet tree in a domain decomposition fashion, using our first algorithm in each segment, reaching $O(\lg n)$ time and $O(n \lg \sigma + p\sigma \lg n / \lg \sigma)$ bits of extra space, where p is the number of available cores. Both algorithms are practical and report good speedup for large real datasets.

Keywords Succinct data structure · Wavelet tree construction · Multicore · Parallel algorithm

A previous version of this paper appeared in the 13th International Symposium on Experimental Algorithms (SEA 2014) [15].

✉ José Fuentes-Sepúlveda
jfuentess@udec.cl

✉ Leo Ferres
lferres@udd.cl

Erick Elejalde
eelejalde@udec.cl

Diego Seco
dseco@udec.cl

[1] Department of Computer Science, Universidad de Concepción, Concepción, Chile

[2] Faculty of Engineering, Universidad del Desarrollo, Santiago, Chile

1 Introduction and motivation

After their introduction in the mid-2000s, *multicore* computers—computers with more than one processing unit (called *cores*) and shared memory—have become pervasive. In fact, nowadays it is difficult to find a single-processor desktop, let alone a high-end server. The argument in favor of multicore systems is simple: Thermodynamic and material considerations prevent chip manufacturers from ever-increasing clock rates. Since 2005, clock frequencies have stagnated at around 3.75GHz for commodity computers, and even in 2015, 4GHz computers are still high end (the Intel Core i7-4790K is a prime example). Thus, the next step in performance is to take advantage of the processing power of multicore computers. In order to do this, algorithms and data structures will have to be modified to make them behave well in these parallel architectures.

At the same time, the amount of data to be processed has become large enough that the ability to maintain it close to the processor is vital. This, in turn, has generated a keen interest in succinct data structures, which besides reducing storage requirements may also reduce the number of memory transfers, the energy consumption, and can be used in low-capacity devices such as smartphones. One such structure that has benefited from thorough research is the *wavelet tree* [18], henceforth *wtree*. Although the *wtree* was originally devised as a data structure for encoding a reordering of the elements of a sequence [14,18], it has now been successfully used in many critical applications such as indexing documents [33], processing grids [25] and sets of rectangles [4], to name but a few. Two excellent surveys have been written about this data structure [22,24], and we refer the readers to them for more application examples and details.

These succinct data structures, however, are generally quite expensive (in time) to build, particularly as the size of the alphabet and the size of the input increase, as is the case nowadays with the so-called big data revolution. We believe parallel computing is a good tool for speeding up the processing of succinct data structures. Unfortunately, (practical) parallel computing suffers from several drawbacks that make these high-performance algorithms difficult to come by: maintaing thread independence while communicating results, keeping clear of "thrashing" the memory hierarchy are two such problems. Thus, a sizeable contribution to the state of the art would involve designing algorithms with good theoretical running times that are also practical in modern commodity architectures with more than one core, which would also help speed up processing of the target data structures in distributed systems: If one node is faster, the whole cluster would be faster as well.

Motivating example Perhaps one of the prominent areas of research in the last few years has been the analysis of genomic data [26,28,30]. In combination with the *Burrows–Wheeler transform* [5], the *wtree* has been used to construct compressed full-text indexes (the *FM-index* [12,13]) over DNA sequences. The structure supports efficient algorithms for important problems in bioinformatics such as the identification of patterns (like the mutations that are known to cause some diseases) or the alignment of short DNA sequences, known as reads (which is a fundamental step to reconstruct a genome), all this *without decompressing the data*. The cost of DNA sequencing has plummeted in the last few years thanks to next-generation sequencing technologies [34]. In addition, these technologies are also much faster. For example, in 2005, a single sequencing run could generate at most one gigabase of data. Meanwhile, in 2014, a single sequencing run could generate up to 1.8 terabases of data [20]. These two factors have drastically increased the amount of genomic data to be processed. Therefore, full-text indexes based on *wtree*s need to be updated periodically. These updates do not modify the already indexed data but add new sequences. This process is not trivial because the *Burrows–Wheeler transform* is a reorganization of the whole sequence in order to make it more compressible. In order to support these updates, there are two options: the use of fully dynamic *wtree*s or the periodic reconstruction of the *wtree* (a solution used in other domains such as Web search engines). Dynamic versions of *wtree*s are quite slow in both update and rank/select operations (see Sect. 3.2 of [22]). The other option is the usage of a static *wtree* and a buffer (which stores the updates since the last reconstruction of the static index). To support queries, both the static *wtree* and the buffer are used. When the buffer is full, the static *wtree* is reconstructed considering the symbols on the buffer, which is emptied after that. Thus, improving the construction time of static *wtree*s becomes critical, for example, to provide solutions in this kind of dynamic domain in which queries are much more frequent than updates.

In this paper, we propose two parallel algorithms for the most expensive operation on *wtree*s: its construction. The first algorithm, `pwt`, has $O(n)$ time complexity and uses $O(n \lg \sigma + \sigma \lg n)$ bits of space,[1] including the space of the final *wtree* and excluding the input, where σ is the size of the alphabet. The second algorithm, `dd`, is an improved version of the `dd` algorithm presented on [15]. This new version has $O(\lg n)$ time complexity and uses $O(n \lg \sigma + p\sigma \lg n / \lg \sigma)$ bits of space, using p threads (see Sect. 3). The `pwt` algorithm improves the $O(n)$ memory consumption in [29]. Meanwhile, the new `dd` algorithm improves the $O(n)$ time complexity of our previous work [15] and the time complexity of [29] by a factor of $O(\lg \sigma)$. We report experiments that demonstrate the algorithms to be not only theoretically good, but also practical for large datasets on commodity architectures, achieving good speedup (Sect. 4). As far as we can tell, we use the largest datasets to date, and our algorithms are faster for most use cases than the state of the art [29].

2 Background and related work

2.1 Dynamic multithreading model

Dynamic multithreading (DYM) [10, Chapter 27] is a model of parallel computation, which is faithful to several industry standards such as Intel's CilkPlus (cilkplus.org), OpenMP Tasks (openmp.org/wp), and Threading Building Blocks (threadingbuildingblocks.org). Besides its mathematical rigor, it is precisely this adoption by many high-end compiler vendors that make the model so appealing for practical parallel algorithms.

In the *Dynamic multithreading* model, a *multithreaded computation* is defined as a directed acyclic graph (DAG) $G = (V, E)$, where the set of vertices V are instructions and $(u, v) \in E$ are dependencies between the instructions, whereby in this case, u must be executed before v.[2] In order to signal parallel execution, we augment sequential pseudocode with three keywords, **spawn**, **sync** and **parfor**. The **spawn** keyword signals that the procedure call that it precedes *may be* executed in parallel with the next instruction in the instance that executes the **spawn**. In turn, the **sync** keyword signals that all spawned procedures must finish before proceeding with the next instruction in the stream. Finally, **parfor** is simply "syntactic sugar" for **spawn**'ing and **sync**'ing ranges of a loop iteration. 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

[1] We use $\lg x = \log_2 x$.

[2] Notice that the RAM model is a subset of the DYM model where the outdegree of every vertex $v \in V$ is ≤ 1.

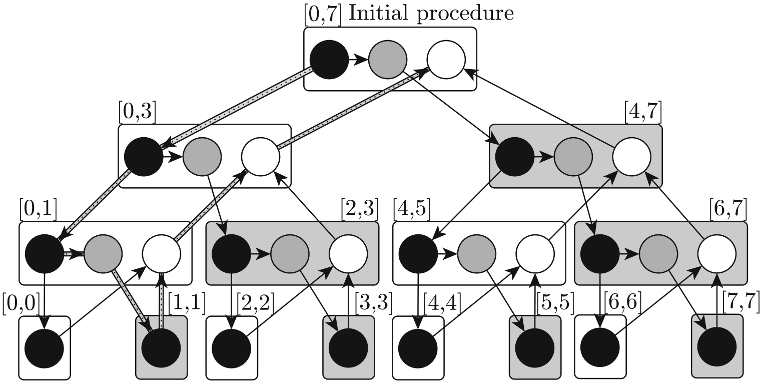


Fig. 1 Example of a multithreaded computation on the dynamic multithreading model. It corresponds to the directed acyclic graph representation of Algorithm 1. *Vertices* represent strands and *edges* represent dependences

these instructions into a single *strand*. The **parfor** keyword, which we use repeatedly here, 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. For example, Algorithm 1 represents a parallel algorithm using **parfor**, and Fig. 1 shows its multithreaded computation. In the figure, each circle represents one strand and each rounded rectangle represents strands that belong to the same procedure call. The algorithm starts on the initial procedure call with the entire range [0, 7]. The first half of the range is **spawn**ed (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 iteration of the loop (black circles on the bottom of the figure). Once an iteration is finished, the corresponding strand **sync**s to its calling procedure (white circles), until reaching the final strand (white circle of the initial call). For more examples of the usage of the DYM model, see [10, Chapter 27]. Strands are scheduled onto cores using a *work-stealing* scheduler, which does the load balancing of the computations. Work-stealing schedulers have been proved to be a factor of 2 away from optimal performance [3].

```
$A$ : array of 8 numbers
parfor $i = 0$ to 7 do
    $A[i] = 0$
return
```

Algorithm 1: Example of a parallel algorithm using the **parfor** keyword. In parallel, the algorithm initializes all the elements of the array A with 0.

To measure the efficiency of our parallel wavelet tree algorithms, we use three metrics: the *work*, the *span* and *speedup*. In accordance with the parallel literature, we will subscript running times by p, so T_p is the running time of an algorithm on p cores. The *work* is the total running time taken by all strands when executing on a *single* core (i.e., T_1), while the *span*, denoted as T_∞, is the *critical path* (the longest path) of G. In Fig. 1, assuming that each strand takes unit time, the work is 29 time units and the span is 8 time units (this is represented in the figure with thicker edges). In this paper, we are interested in speeding up wavelet tree manipulation and improving the lower bounds of this speedup. To measure this, we will define *speedup* as $T_1/T_P = O(p)$, where linear speedup $T_1/T_p = \Theta(p)$ is the goal and the theoretical upper bound. We also define *parallelism* as the ratio T_1/T_∞, the theoretical maximum number of cores for which it is possible to achieve linear speedup.

2.2 Wavelet trees

A wavelet tree (*wtree*) is a data structure that maintains a sequence of n symbols $S = s_1, s_2, \ldots, s_n$ over an alphabet $\Sigma = [1..\sigma]$ under the following operations: $access(S, i)$, which returns the symbol at position i in S; $rank_c(S, i)$, which counts the times symbol c appears up to position i in S; and $select_c(S, j)$, which returns the position in S of the j-th appearance of symbol c. Storage space of *wtree*s can be bounded by different measures of the entropy of the underlying data, thus enabling compression. In addition, they can be implemented efficiently [7] and perform well in practice.

The *wtree* is a balanced binary tree. We identify the two children of a node as left and right. Each node represents a range $R \subseteq [1, \sigma]$ of the alphabet Σ, its left child represents a subset R_l, which corresponds with the first half of R, and its right child a subset R_r, which corresponds with the second half. Every node virtually represents a subsequence S' of S composed of symbols whose value lies in R. This subsequence is stored as a bitmap in which a 0 bit means that position i belongs to R_l and a 1 bit means that it belongs to R_r.

At its simplest, a *wtree* requires $n\lceil \lg \sigma \rceil + o(n \lg \sigma)$ bits for the data, plus $O(\sigma \lg n)$ bits to store the topology of the tree (considering one pointer per node), and supports aforementioned queries in $O(\lg \sigma)$ time by traversing the tree using $O(1)$-time *rank*/*select* operations on bitmaps [27]. A simple recursive construction algorithm takes $O(n \lg \sigma)$ time. As mentioned before, the space required by the structure can be reduced: The data can be compressed and stored in space bounded by its entropy (via compressed encodings of bitmaps and modifications on the shape of the tree), and the $O(\sigma \lg n)$ bits of the topology can be removed, effectively using one pointer per *level* of the tree [7], which is important for large alphabets. We focus on construction using a pointer per level because, even though it adds some running time costs, it is more suitable for *big data*. This notwithstanding, it is trivial to apply the technique to the one-pointer-per-node construction case, and our results can be readily extended to other encodings and tree shapes.

Figure 2 shows an example of two *wtree* representations for the sequence $S =$ "once upon a time a PhD student." Figure 2a shows the one-pointer-per-node representation, while Fig. 2b shows the one-pointer-per-level representation. In our algorithms, we implemented the one-pointer-per-level representation; however, for clarity, we use the one-pointer-per-node representation to exemplify. In both representations, we highlighted the traversal performed by the operation $access(S, 24)$. To answer it, a top-down traversal of the *wtree* is performed: If a bit 0 is found, we visit the left branch; if a 1, the right branch is chosen. In the first representation, the query works as follows: Let *curr* be the root, B_{curr} be the bitmap of the current node, $i = 24$ be the index of interest, R be the range $[0, \sigma - 1] = [0, 15]$ and $rank_c(B_{curr}, i)$ be the number of c-bits up to position i in B_{curr}. At the beginning, we inspect the bit $B_{curr}[i]$. Since the bit is 1, we recompute $i = rank_1(B_{curr}, i) - 1 = 7$, change *curr* to be the right child of *curr* and halve $R = [8, 15]$. Then, we repeat the process. Since $B_{curr}[i] = 0$, $i = rank_0(B_{curr}, i) - 1 = 4$, *curr* is updated to be the left child of *curr* and $R = [8, 11]$. Now, $B_{curr}[i] = 0$, $i = rank_0(B_{curr}, i) - 1 = 2$, *curr* is changed to be the left child of *curr* and $R = [8, 9]$. Finally, in the last level, $B_{curr}[i] = 0$, so the range $R = [8, 8]$ and the answer for $access(S, 24)$ is $\Sigma[8] =$ 't'. $rank_c(S, i)$ and $select_c(S, i)$ perform similar traversals to $access(S, i)$. For a more detailed explanation of *wtree* operations, see [24]. For

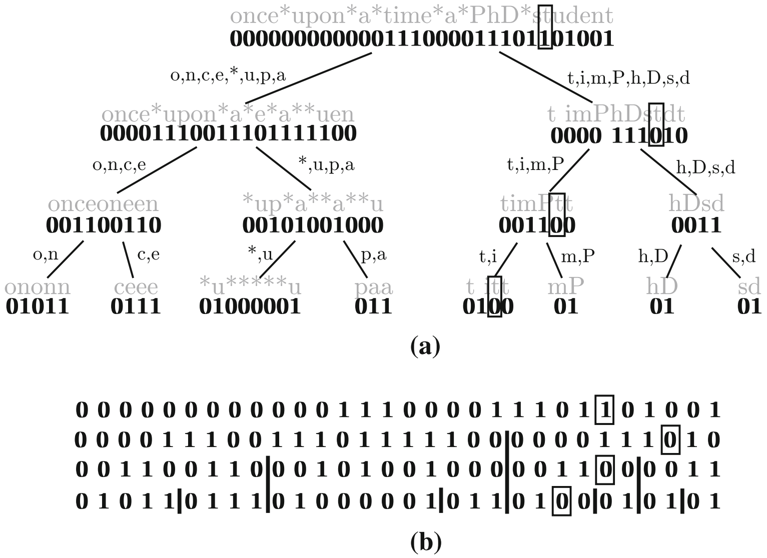


Fig. 2 A *wtree* for the sequence S = "once upon a time a PhD student" and the contiguous alphabet $\Sigma = \{$o, n, c, e,", u, p, a, t, i, m, P, h, D, s, d$\}$. We draw spaces using *stars*. **a** Representation of a *wtree* using one pointer per node and its associated bitmap. The subsequences of S in the nodes (*gray font*) and the subsets of Σ in the edges are drawn for illustration purposes. **b** Representation of a *wtree* using one pointer per level and its associated n-bit bitmap. It can simulate the navigation on the tree by using the rank operation over the bitmaps

the one-pointer-*per-level* representation, the procedure is similar, with the exception that the traversal of the tree must be simulated with rank operations over the bitmaps [7].

Practical implementations of *wtree*s can be found in LIBCDS [6] and SDSL [16]. LIBCDS implements a recursive construction algorithm that works by halving Σ into binary sub-trees whose left children are all 0s and right children are all 1s, until 1s and 0s mean only one symbol in Σ. SDSL implements an algorithm based on the idea of *counting sort* that is more efficient in memory. The algorithm counts the number of bits that will be placed in each node of the *wtree*, computing the position of each symbol in each level of the *wtree*, which avoids maintaining a permutation of the input. Both libraries are the best current sequential implementations available, without considering space-efficient construction algorithms [9,31].

As of late, some work has been done in parallel processing of *wtree*s. In [1], the authors explore the use of wavelet trees in Web search engines. They assume a distributed memory model and propose partition techniques to balance the workload of processing *wtree*s. Note that our work is complementary to theirs, as each node in their distributed system can be assumed to be a multicore computer that can benefit from our algorithms. In [21], the authors explore the use of SIMD instructions to improve the performance of *wtree*s and other string algorithms [11]. This set of instructions can be considered as low-level parallelism, since they use instructions in modern processors that work by joining registers for some integer computation, dealing with 128-bit integers at a time. We can also benefit from their work as it may improve the performance of the sequential parts of our algorithms. However, we leave this optimization for future work.

In [15], we introduced the first two parallel algorithms for *wtree* construction: pwt and dd, both with $O(n)$ time complexity. The details of pwt and an improvement in dd are given in Sects. 3.1 and 3.2, respectively. Based on [15], Shun [29] introduces two new parallel algorithms. The first algorithm, called `levelWT`, constructs the *wtree* level by level. In each of the $\lceil \lg \sigma \rceil$ levels, the algorithm uses a parallel prefix sum algorithm to compute the position of the bits, constructing the nodes and their bitmaps in parallel with $O(n)$ work and $O(\lg n)$ span, which results in $O(n \lg \sigma)$ work and $O(\lg n \lg \sigma)$ span. The second algorithm, called `sortWT`, constructs all levels in parallel, similar to our original `pwt`, instead of one by one. For a level l, the `sortWT` algorithm applies a parallel stable integer sorting using the l most significant bits of each symbol as the key. With the sorted input sequence, the algorithm fills the corresponding bitarrays in parallel, using parallel prefix sum and filter algorithms to compute the position of the bits. The total work of the `sortWT` algorithm is $O(W_{sort} \lg \sigma)$, where W_{sort} is the work incurred by sorting, and $O(S_{sort} + \lg n)$ is the span, and where, in turn, S_{sort} corresponds to the span of the sorting algorithm and the $\lg n$ component is the span of the prefix sum and filter algorithms. The author also discusses a variation of the `sortWT` algorithm, reaching $O(n \lg \sigma)$ work and $O(\lg n \lg \sigma)$ span. In practice, the `levelWT` algorithm shows better performance. Compared to our previous algorithms, the `levelWT` and `sortWT` algorithms can scale beyond $O(\lg \sigma)$ cores. However, both also need to duplicate and modify the input sequence, resulting in an increase in memory usage, requiring $O(n \lg n)$ bits of extra space.

2.3 Problem statement

The *wtree* is a versatile data structure that uses $n \lg \sigma + o(n \lg \sigma)$ bits of space and supports several queries (such as access, rank and select) in $O(\lg \sigma)$ time, for a sequence of n symbols over an alphabet Σ of size σ. The *wtree* can be constructed in $O(n \lg \sigma)$ time, which may be prohibitive for large sequences. Therefore, in this work, we reduce the time complexity of the most time-consuming operation of *wtree*, its construction, on multicore architectures. Given a multicore machine with p available cores, we propose the design and implementation of parallel algorithms to the construction of *wtree*. The proposed algorithms scale with p, achieving good practical speedups and extra memory usage.

3 Multicore wavelet tree

We focus on binary wavelet trees where the symbols in Σ are contiguous in $[1, \sigma]$. If they are not contiguous, a bitmap is used to remap the sequence to a contiguous alphabet [7]. Under these restrictions, the *wtree* is a balanced binary tree with $\lceil \lg \sigma \rceil$ levels. In this section, we build the representation of *wtree*s that removes the $O(\sigma \lg n)$ bits of the topology. Hence, when we refer to a *node*, this is a conceptual node that does not exist in the actual implementation of the data structure.

In what follows, two iterative construction algorithms are introduced that capitalize on the idea that any level of the *wtree* can be built independently from the others. Unlike in classical *wtree* construction, when building a level we cannot assume that any previous step is providing us with the correct permutation of the elements of S. Instead, we compute the node at level i for each symbol of the original sequence. More formally,

Proposition 1 *Given a symbol $s \in S$ and a level i, $0 \leq i < l = \lceil \lg \sigma \rceil$, of a wtree, the node at which s is represented at level i can be computed as $s \gg l - i$.*

In other words, if the symbols of Σ are contiguous, then the i most significant bits of the symbol s give us its corresponding node at level i. In the word-RAM model with word size

```
   Input  : $S, n, \sigma$
   Output: A wavelet tree representation $WT$ of $S$
1  $WT$ is a new wavelet tree with $\lceil \lg \sigma \rceil$ levels
2  parfor $i = 0$ to $\lceil \lg \sigma \rceil - 1$ do
3      $B$ is a bitarray of size $n$
4      $C$ is an integer array of size $2^i$
5      for $j = 0$ to $n - 1$ do
6          increment($C[S[j]/2^{\lceil \lg \sigma \rceil - i}]$)
7      parPrefixSum($C$)
8      for $j = 0$ to $n - 1$ do
9          if $(S[j]\ \&\ 2^{\lceil \lg \sigma \rceil - i - 1}) == 1$ then
10             bitmapSetBit($B, C[S[j]/2^{\lceil \lg \sigma \rceil - i}], 1$)
11         else
12             bitmapSetBit($B, C[S[j]/2^{\lceil \lg \sigma \rceil - i}], 0$)
13         increment($C[S[j]/2^{\lceil \lg \sigma \rceil - i}]$)
14     $WT[i]$ = createRankSelect($B$)
15 return $WT$
```

Algorithm 2: Per-level parallel algorithm (`pwt`)

$\Omega(\lg n)$, this computation takes $O(1)$ time. Since the word-RAM model is a subset of the DYM model[2], the following corollary holds:

Corollary 1 *The node at which a symbol s is represented at level i can be computed in $O(1)$ time.*

3.1 Per-level parallel algorithm

Our first algorithm, called `pwt`, is shown in Algorithm 2 (the sequential version can be obtained by replacing **parfor** instructions with sequential **for** instructions). The algorithm takes as input a sequence of symbols S, the length n of S, and the length of the alphabet, σ (see Sect. 2.2). The output is a *wtree* WT that represents S. We denote the ith level of WT as $WT[i]$, $\forall i, 0 \leq i < \lceil \lg \sigma \rceil$.

The outer loop (line 2) iterates in parallel over $\lceil \lg \sigma \rceil$ levels. Lines 3 to 14 scan each level performing the following tasks: The first step (lines 3 and 4) initializes the bitmap B of the ith level and initializes an array of integers C. The array C will be used to count the number of bits in each node of the *wtree* at level i, using *counting sort*. The second step (lines 5 and 6) computes the size of each node in the ith level performing a linear-time sweep over S. For each symbol in S, the algorithm computes the corresponding node for alphabet range at the current level. The expression $S[j]/2^{\lceil \lg \sigma \rceil - i}$ in line 6 shows an equivalent representation of the idea in Proposition 1. The third step performs a parallel prefix sum algorithm [19] over the array C, obtaining the offset of each node. Once the offset of the nodes is known, the algorithm constructs the corresponding bitarray B, sequentially scanning S (lines 8 to 13). For each symbol in S, the algorithm computes the corresponding node and whether the symbol belongs to either the first or second half of Σ for that node. The corresponding bit is set using *bitmapSetBit* at position $C[S[j]/2^{\lceil \lg \sigma \rceil - i}]$. Line 14 creates the rank/select structures of the bitmap B of the ith level.

Figure 3 shows an snapshot of the execution of the `pwt` for the input sequence of Fig. 2: The levels of the *wtree* can be constructed in different threads asynchronously.

The work T_1 of this algorithm takes $O(n \lg \sigma)$ time. This matches the time for construction found in the literature. Each of the $\lg \sigma$ tasks that create the `pwt` algorithm has a complexity of $O(n + \sigma/p + \lg p)$, due to the scans over the input sequence and the parallel prefix sum

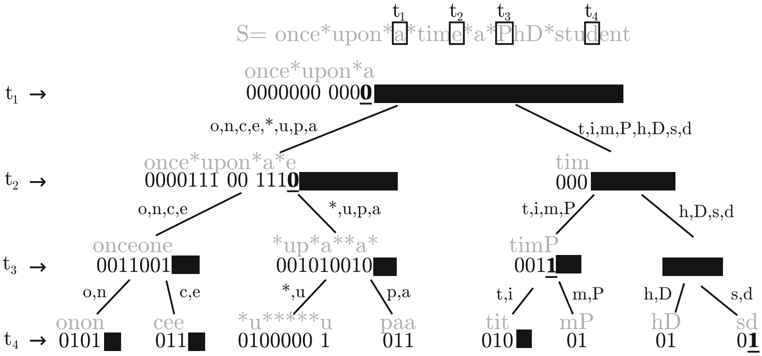


Fig. 3 Snapshot of an execution of the algorithm pwt for the sequence introduced in Fig. 2. In the snapshot, thread t_1 is writing the first bit of the symbol $S[10]$ = 'a' at level 0, thread t_2 is writing the second bit of $S[15]$ = 'e' at level 1, thread t_3 is writing the third bit of $S[19]$ = 'P' at level 2 and thread t_4 is writing the fourth bit of $S[26]$ = 'd' at level 3. *Black areas* represent bits associated with unprocessed symbols

over the array C. The work of pwt is still $T_1 = O(n \lg \sigma)$. Since all tasks have the same complexity, assuming constant access to any position in memory, the critical path is given by the construction of one level of the *wtree*. That is, for $p = \infty$, $T_\infty = O(n + \lg \sigma) = O(n)$. In the same vein, parallelism will be $T_1/T_\infty = O(\lg \sigma)$. It follows that having $p \leq \lg \sigma$ the algorithm will obtain optimal speedup. The overhead added for the **parfor**, $O(\lg \lg \sigma)$ is negligible. With respect to the working space, the algorithm pwt needs the space of the *wtree* and the extra space for the array C, that is, a working space of $O(n \lg \sigma + \sigma \lg n)$ bits.

The main drawback of the pwt algorithm is that it only scales linearly until the number of cores equals the number of levels in the wavelet tree. So, even if we have more cores available, the algorithm will only use efficiently up to $\lg \sigma$ cores. Nevertheless, this algorithm is simple to implement and suitable in domains where there is not possible to use all available resources to the construction of *wtree*s.

3.2 Domain decomposition parallel algorithm

The second algorithm that we propose makes efficient use of all available cores. The main idea of the algorithm is to divide the input sequence S in $k = O(p/\lg(\sigma))$ segments of size $O(n/k)$ and then apply the pwt algorithm on each segment, generating $O(\lg \sigma)$ tasks per segment and creating k partial *wtree*s. After that, the algorithm merges all the partial *wtree*s into a single one that represents the entire input text. We call this algorithm dd because of its domain decomposition nature. This algorithm improves the $O(n)$ time complexity of the one introduced previously in [15].

The dd algorithm is shown in Algorithm 3. It takes the same input as pwt with the addition of the number of segments, k. The output is a *wtree* WT, which represents the input data S.

The first step of dd (lines 1–4) allocates memory for the output *wtree*, its bitarrays, B, the bitarrays of the partial *wtree*s, pB, and two 3-dimensional arrays of numbers, L and G, where the third dimension changes according to the number of nodes in each level. Arrays L and G store local and global offsets, respectively. The local offsets store the offsets of all the nodes of the partial *wtree*s with respect to the partial *wtree* containing them. Similarly, G stores the offsets of all the nodes of the partial *wtree*s with respect to the final *wtree*. In other words, each entry $L[a][b][c]$ stores the position of node c at level b whose parent is partial *wtree* a. Each entry $G[a][b][c]$ stores the position of node c at level b in the partial *wtree* a

```
   Input  : $S, n, \sigma, k$
   Output: A wavelet tree representation $WT$ of $S$
1  $WT$ is a new tree with $\lceil \lg \sigma \rceil$ levels
2  $B$ is an array of $\lceil \lg \sigma \rceil$ bitarrays of size $n$
3  $pB$ is a bidimensional array of bitarrays of dimensions $k \times \lceil \lg \sigma \rceil$
4  $G, L$ are tridimensional arrays of integers of dimensions $k \times \lceil \lg \sigma \rceil \times 2^{level}$
5  parfor $i = 0$ to $k - 1$ do
6      $pB[i] =$ createPartialBA($S, \sigma, i, n/k$)
7  parfor $i = 0$ to $\lceil \lg \sigma \rceil - 1$ do
8      parPrefixSum($i, k$)
9  $B =$ mergeBA($n, \sigma, k, pB$)
10 parfor $i = 0$ to $\lceil \lg \sigma \rceil - 1$ do
11     $WT[i] =$ createRankSelect($B[i]$)
12 return $WT$
```

Algorithm 3: Domain decomposition parallel algorithm (dd)

```
   Input  : $S, \sigma, k', n$
   Output: A bitarray representation $B$ of the $k'$th segment of $S$
1  $B$ is an array of $\lceil \lg \sigma \rceil$ bitarrays of size $n$
2  parfor $i = 0$ to $\lceil \lg \sigma \rceil - 1$ do
3      for $j = n \times k'$ to $n \times (k' + 1) - 1$ do
4          increment($G[k'][i][S[j]/2^{\lceil \lg \sigma \rceil - i}]$)
5      prefixSum($G, L$)
6      for $j = n \times k'$ to $n \times (k' + 1) - 1$ do
7          if $(S[j]$ & $2^{\lceil \lg \sigma \rceil - i - 1}) == 1$ then
8              bitmapSetBit($B, G[k'][i][S[j]/2^{\lceil \lg \sigma \rceil - i}], 1$)
9          else
10             bitmapSetBit($B, G[k'][i][S[j]/2^{\lceil \lg \sigma \rceil - i}], 0$)
11         increment($G[k'][i][S[j]/2^{\lceil \lg \sigma \rceil - i}]$)
12 return $B$
```

Function createPartialBA

```
   Input  : $n, \sigma, k, pB$
   Output: A bitarray representation $B$ of the input sequence $S$
1  $B$ is an array of $\lceil \lg \sigma \rceil$ bitarrays of size $n$
2  parfor $i = 0$ to $\lceil \lg \sigma \rceil - 1$ do
3      parfor $j = 0$ to $k - 1$ do
4          parfor $m = j \times 2^i$ to $(j + 1) \times 2^i$ do
5              $dst = B[i]$              // Destination of the bits to be copied
6              $src = pB[m \bmod k][i]$        // Source of the bits to be copied
7              $go = G[m \bmod k][i][m/k]$                  // Offset in $dst$
8              $lo = L[m \bmod k][i][m/k]$                  // Offset in $src$
9              $nb = L[m \bmod k][i][m/k + 1] - L[m \bmod k][i][m/k]$     // Number of bits
10             parallelBitarrayConcat($dest, src, go, lo, nb$)
11 return $B$
```

Function mergeBA

inside the final *wtree*. We will treat the arrays L and G as global variables to simplify the pseudocode.

The second step (lines 5 and 6) computes the partial *wtree*s of the k segments in parallel. For each segment, createPartialBA is called to create the partial *wtree*. This function is similar to the one in the pwt algorithm, performing a prefix sum (line 5 in Function createPartialBA) to compute the local offsets and store them both in G and L. We reuse the array G to save

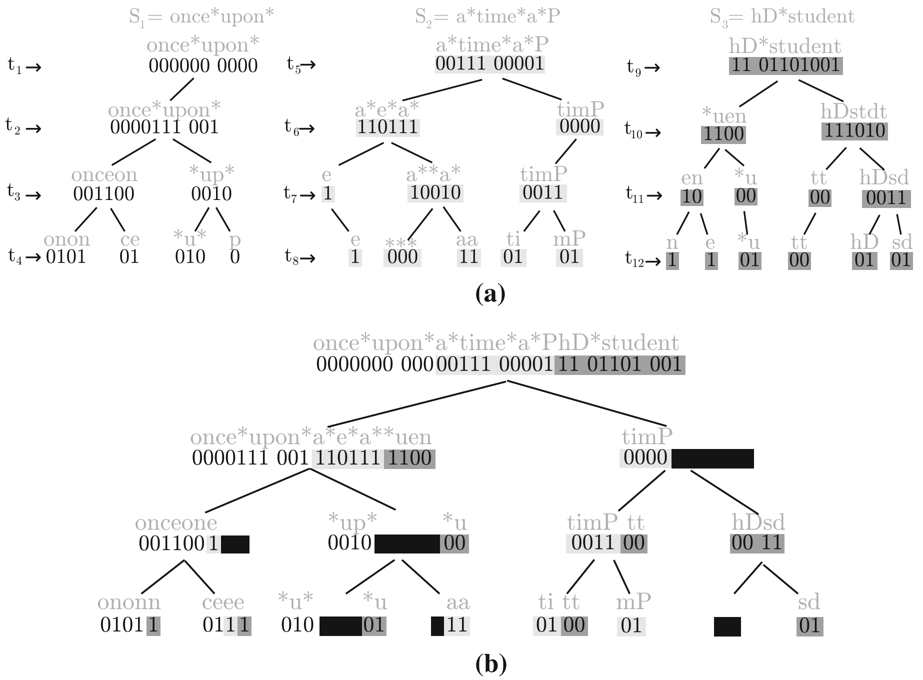


Fig. 4 Snapshot of an execution of the algorithm dd. Figure 4**a, b** represents snapshots of Functions createPartialBA and mergeBA, respectively. The result of this example is the *wtree* of Fig. 2a. **a** Snapshot of Function createPartialBA. The figure shows the construction of the partial *wtree*s after the split of the input sequence introduced in Fig. 2 into three subsequences. To create each partial *wtree*, the algorithm uses the pwt algorithm. These partial *wtree*s are the input of Function mergeBA. **b** Snapshot of the Function mergeBA. *White*, *light gray* and *dark gray* bitarrays represent the bitarrays of first, second and third partial *wtree*s, respectively. The positions of the partial *wtree*s bitarrays are computed in advance; therefore, such bitarrays can be copied to the final *wtree* in parallel. *Black areas* represent uncopied bits

memory in the next step. Notice that the output of the function is a partial *wtree* composed of $\lceil \lg \sigma \rceil$ bitarrays, without rank/select structures over such bitarrays.

The third step of the dd algorithm uses the local offsets stored in L to compute the global ones (lines 7 and 8). To do that, at each level i, the algorithm applies a parallel prefix sum algorithm using the k local offsets of that level. The prefix sum algorithm uses the implicit total order within the local offsets. Since each level in the offsets is independent of the others, we can apply the $\lceil \lg \sigma \rceil$ calls of the parallel prefix sum algorithm in parallel.

Once we have the global offsets computed, the fourth step merges all partial *wtree*s, in parallel. Function mergeBA creates one parallel task for each node in the partial *wtree*s. In each parallel task (lines 5 to 10) the function concatenates the bitarray of the node m/k of the ith level of the m *mod* k partial *wtree* into the corresponding bitarray, $B[i]$, of the final *wtree*. Using the local and the global offsets, the function parallelBitarrayConcat copies the *nb* of $pB[i]$, starting at position $L[m \textit{ mod } k][i][m/k]$ into the bitarray $B[i]$ at position $G[m \textit{ mod } k][i][m/k]$. The function parallelBitarrayConcat is *thread safe*: The first and last machine words that compose each bitarray are copied using atomic operations. Thus, the concatenated bitarrays are correct regardless of multiple concurrent concatenations. The last step, lines 10–11, creates the rank/select structures for each level of the final *wtree*.

For an example of the algorithm, see Fig. 4. Figure 4a shows a snapshot of the function createPartialBA, and Fig. 4b shows a snapshot of mergeBA

The dd algorithm has the same asymptotic complexity as pwt, with work $T_1 = O(n \lg \sigma)$. When running on p cores and dividing S in $k = O(p/\lg \sigma)$ segments, the construction of the partial *wtree*s takes $O(n \lg \sigma / p)$ time. The prefix sum takes $O(\sigma / \lg \sigma + \lg p)$ time [19]. Merge takes $O(n \lg \sigma / pw)$, where w is the word size of that architecture. The overhead of the **parfor**s is $O(\lg p + \lg \sigma \lg \lg \sigma)$. For $p = \infty$, the span of the construction of the partial *wtree*s is $O(1)$, $O(\lg(k\sigma))$ for the prefix sum section and $O(1)$ for the merge function. In the case of the merge function, the offsets of the bitarrays have been previously computed and each bit can be copied in parallel. Thus, considering w as a constant and $k = O(p/\lg \sigma)$, the span is $T_\infty = O(\lg n)$ in all cases.

The working space needed by dd is limited by the space needed for the *wtree*, the partial *wtree*s, and local and global offsets, totaling $O(n \lg \sigma + k\sigma \lg n)$ bits. By manipulating the value of k, however, we can reduce the needed space or improve the performance of dd algorithm. If $k = 1$, then space is reduced to $O(n \lg \sigma + \sigma \lg n)$ bits, but this limits scalability to $p < \lg \sigma$. If $k = p$, we improve the time complexity, at the cost of $O(n \lg \sigma + p\sigma \lg n)$ bits.

4 Experimental evaluation

We tested the implementation of our parallel wavelet tree construction algorithms considering one pointer per level and without considering the construction time of rank/select structures. We compared our algorithms against LIBCDS,[3] SDSL and the fastest algorithm in [29]. Both libraries were compiled with their default options and the -O2 optimization flag. With regard to the bitarray implementation, we use the 5%-extra space structure presented in [17] (as LIBCDS does). For SDSL, we use the bit_vector implementation with settings rank_support_scan<1>, select_support_scan<1> and select_support_scan<0> to skip construction time of rank/select structures. In our experiments, shun is the fastest of the three algorithms introduced in [29], compiled also with the -O2 optimization flag. Our dd algorithm was tested with $k = p$ privileging time performance over memory.

4.1 Experimental setup

All algorithms were implemented in the C programming language and compiled with GCC 4.9 (Cilk branch) using the -O2 optimization flag. The experiments were carried out on a 4-chip (8 NUMA nodes) AMD Opteron$^{\text{TM}}$ 6278 machine with 8 physical cores per NUMA node, clocking at 2.4 GHz each, with one 64KB L1 instruction cache shared by two cores, one 16 KB L1 data cache per core, a 2 MB L2 cache shared between two cores, and a 6 MB of L3 shared between 8 cores per NUMA node. The machine had 192 GB of DDR3 RAM, clocking at 1333 MHz, with 24 GB per NUMA node. Algorithms were compared in terms of running times using the usual high-resolution (nanosecond) C functions in <time.h>. Memory usage was measured using the tools provided by malloc_count [2].

The experimental trials consisted in running the algorithms on datasets of different alphabet sizes, input sizes n and number of cores. The datasets are listed in Table 1. We distinguish between two types of datasets: those in which symbols are encoded using 1 byte, and those

[3] We also tested a new version of LIBCDS called LIBCDS2; however, the former had better running times for the construction of *wtree*s.

Table 1 Datasets used in the experiments

	Dataset	n	σ
1	rna.512MB	536,870,912	4
2	rna.1GB	1, 073, 741, 824	4
3	rna.2GB	2,147,483, 648	4
4	rna.3GB	3,221,225,472	4
5	rna.4GB	4,294,967,296	4
6	rna.5GB	5,368,709,120	4
7	rna.6GB	6,442,450,944	4
8	rna.13GB	14,570,010,837	4
9	prot	1,184,051,855	27
10	src.200MB	210,866,607	230
11	src.98MB	25,910,717	2,446,383
12	src.512MB	134,217,728	2,446,383
13	src.1GB	268,435,455	2,446,383
14	src.2GB	536,870,911	2,446,383
15	en.x.27	134,217,728	2^x
16	en.x.28	268,435,456	2^x
17	en.x.29	536,870,912	2^x
18	en.x.30	1,073,741,824	2^x

In the datasets 15–18, x can take values in $\{4, 6, 8, 10, 12, 14\}$

in which symbols are encoded using 4 bytes. Datasets 1–10 in Table 1 with $\sigma \leq 256$ were encoded using 1 byte. Datasets 11–14 were encoded using 4 bytes. Datasets 15–18, that have $\sigma = 2^x$, were encoded as follows: For $x = \{4, 6, 8\}$, symbols were encoded with a single byte. For $x = \{10, 12, 14\}$, symbols were encoded in four bytes. The dataset `rna.13GB` is the GenBank mRNAs of the University of California, Santa Cruz.[4] The rest of the `rna` datasets were generated by splitting the previous one. We also tested datasets of protein sequences, `prot`[5] and source code, `src.200MB`.[6] We also built a version of the source code dataset using words as symbols, `src.98MB`. The rest of the `src` datasets were generated by concatenating the previous one up to a maximum of 2 GB. To measure the impact of varying the alphabet size, we took the English corpus of the Pizza & Chili website[7] as a sequence of *words* and filtered the number of different symbols in the dataset. The dataset had an initial alphabet Σ of $\sigma = 633{,}816$ symbols. For experimentation, we generated an alphabet Σ' of size 2^x, taking the top 2^x *most frequent* words in the original Σ and then assigning a random index to each symbol using a Marsenne Twister [23], with $x \in \{4, 6, 8, 10, 12, 14\}$. To create an input sequence S of n symbols for the English dataset (`en`), we searched for each symbol in Σ' in the original English text and, when found, appended it to S until it reached the maximum possible size given σ' (∼1.5 GB, in the case of $\sigma' = 2^{18}$), maintaining the order of the original English text. We then either split S until we reached the target size $n = 2^{27}$ or

[4] http://hgdownload.cse.ucsc.edu/goldenPath/hg38/bigZips/xenoMrna.fa.gz (April, 2015).

[5] http://pizzachili.dcc.uchile.cl/texts/protein/proteins.gz (April, 2015).

[6] http://pizzachili.dcc.uchile.cl/texts/code/sources.gz (April, 2015).

[7] http://pizzachili.dcc.uchile.cl/texts/nlang/english.1024MB.gz (March, 2013).

concatenated S with initial subsequences of itself to reach the larger sizes 2^{28}, 2^{29} and 2^{30}. We repeated each trial five times and recorded the median time [32].[8]

4.2 Running times and speedup

Table 2 shows the running times of all tested algorithms.[9] LIBCDS and shun work just for $n < 2^{32}$, so we cannot report running times of these algorithms for the datasets rna.4GB, rna.5GB, rna.6GB and rna.13GB.

For each dataset, we underline the best sequential running times. We use those values to compute speedups. The best parallel times for $p = 64$ are identified using a bold typeface. Although libcds and sdsl are the state of the art in sequential implementations of *wtree*s, the best sequential running times were obtained from the parallel implementations running on one thread. The main reason for this is that SDSL implements a semi-external algorithm for *wtree* construction, involving heavy disk access, while LIBCDS uses a recursive algorithm, with known memory and executions costs.

Figure 5 shows speedups for rna.3GB, prot, src.200MB, en.4.30, src.2GB, en.14.30 datasets, with the largest n. As expected, the pwt algorithm is competitive until $p < \lg \sigma$. Thus, for small σ the pwt algorithm is not the best alternative as shown in Fig. 5a, b, d. If the algorithm recruits more threads than levels, the overhead of handling these threads increases, generating some "noise" in the times obtained. The performance of pwt will be dominated also by the thread that builds more levels. For instance, in Fig. 5f we created a *wtree* with 14 levels. In the case of one thread, that thread has to build the 14 levels. In the case of 4 threads, each has to build three levels. For 8 and 12 threads, some threads will build two levels, so those threads dominate the running time. Finally, for the case of 16 threads, each thread has to build at most one level. This explains the "staircase" effect seen for pwt in Fig. 5f.

In all datasets shown in Fig. 5, except for Fig. 5e, the dd algorithm has a better speedup than both pwt and shun, especially for datasets with small alphabets, such as rna, prot and en.4. In the case of Fig. 5e, shun has a better speedup, because our algorithms have worse data locality, we come back to the impact of locality of reference. It is important to remember that although shun has a better speedup, its memory consumption is larger than in our algorithms, as can be seen in Sect. 4.3.

4.3 Memory consumption

Figure 6 shows the amount of memory allocated with *malloc* and released with *free*. For all algorithms, we report the peak of memory allocation and only considered memory allocated during construction, not memory allocated to store the input text. The datasets are ordered incrementally by n. In the case of the dd algorithm, the figure shows memory consumption for $k = 1$. LIBCDS and shun use more memory during construction time. In fact, pwt uses up to 33 and 25 times less memory than LIBCDS and shun, respectively. Memory usage in libcds is dominated by its recursive nature, while shun copies the input sequence S, of $O(n \lg n)$ bits, to preserve it and to maintain permutations of it in each iteration. Additionally, shun uses an array of size $O(\sigma \lg n)$ bits to maintain some values associated with the nodes

[8] In order to be less sensitive to outliers, we use the median time instead of other statistics. In our experiments, the pwt algorithm showed a larger deviation with respect to the number of threads than the other algorithms. However, the differences were not statistically significant.

[9] A complete report of running times and everything needed to replicate these results is available at www.inf.udec.cl/~josefuentes/wavelettree.

Table 2 Running times, in seconds, of the sequential algorithms and parallel algorithms with 1 and 64 threads

Datasets	`libcds`	`sdsl`	`pwt`		`dd`		`shun`	
			1	64	1	64	1	64
rna.512MB	23.42	32.41	11.83	7.00	12.65	**0.40**	12.63	0.67
rna.1GB	47.38	65.30	23.89	16.19	25.30	**0.62**	25.36	1.32
rna.2GB	100.13	131.86	46.98	27.62	50.80	**1.20**	50.89	2.64
rna.3GB	142.90	220.11	71.09	41.00	75.37	**2.17**	66.35	3.79
rna.4GB	–	198.10	94.39	55.04	101.44	**2.84**	–	–
rna.5GB	–	329.27	117.13	68.24	126.66	**3.57**	–	–
rna.6GB	–	389.25	141.59	81.80	152.57	**4.35**	–	–
rna.13GB	–	881.41	314.86	330.44	333.14	**10.75**	–	–
prot	104.40	142.67	58.54	21.81	68.19	**2.17**	64.06	3.54
src.200MB	24.81	31.41	14.68	2.67	17.70	**0.52**	16.73	1.06
src.98MB	7.92	9.52	5.28	0.77	5.73	3.94	5.07	**0.75**
src.512MB	37.77	49.21	28.94	5.07	28.98	5.36	25.52	**3.07**
src.1GB	75.48	99.95	57.99	8.87	55.36	9.60	49.52	**6.17**
src.2GB	150.67	205.41	112.78	25.30	110.83	15.11	98.11	**11.77**
en.4.27	8.78	14.24	5.75	1.82	6.50	**0.28**	6.98	0.38
en.4.28	15.82	28.53	11.44	3.67	12.88	**0.40**	12.34	0.77
en.4.29	35.43	57.11	23.01	7.22	25.51	**0.84**	24.68	1.57
en.4.30	70.00	113.88	46.10	14.40	51.06	**1.63**	55.56	3.06
en.6.27	12.44	19.10	7.98	1.78	9.58	**0.36**	10.46	0.61
en.6.28	22.65	38.37	15.92	3.33	19.35	**0.52**	18.38	1.17
en.6.29	50.28	76.91	31.78	7.08	37.90	**1.18**	41.86	2.36
en.6.30	99.66	153.72	63.62	15.90	76.59	**2.20**	83.29	4.68
en.8.27	15.87	26.00	11.48	1.87	13.15	**0.46**	14.10	0.88
en.8.28	29.06	52.15	22.86	3.71	26.52	**0.78**	28.28	1.58
en.8.29	64.84	105.01	45.79	7.57	52.53	**1.56**	56.68	3.14
en.8.30	128.65	209.54	91.83	14.65	105.00	**3.13**	113.13	6.26
en.10.27	21.32	33.25	14.61	2.26	13.94	1.66	17.26	**1.39**
en.10.28	43.55	68.00	30.32	6.43	29.05	**2.18**	33.15	2.78
en.10.29	89.96	136.67	60.69	9.25	58.55	**4.59**	67.16	5.67
en.10.30	183.57	281.53	123.88	17.70	119.14	**8.93**	214.2	10.77
en.12.27	24.38	39.09	17.97	2.52	17.33	2.61	20.33	**1.64**
en.12.28	50.17	80.22	37.66	7.62	36.36	**2.66**	38.97	3.25
en.12.29	103.39	161.96	75.09	10.41	72.46	**5.73**	128.35	6.71
en.12.30	211.66	333.32	150.02	20.33	145.04	**9.66**	259.21	12.99
en.14.27	27.44	43.61	21.92	3.10	21.39	2.43	22.51	**1.84**
en.14.28	56.44	90.05	45.85	6.11	44.70	**2.94**	44.53	3.67
en.14.29	116.15	182.46	90.41	12.50	88.37	**6.97**	91.53	7.79
en.14.30	238.36	377.77	184.83	22.31	178.58	**10.50**	302.14	15.98

The best sequential times are underlined, and the best parallel times are shown using bold typeface. A "–" is shown for implementations that just work for $n < 2^{32}$

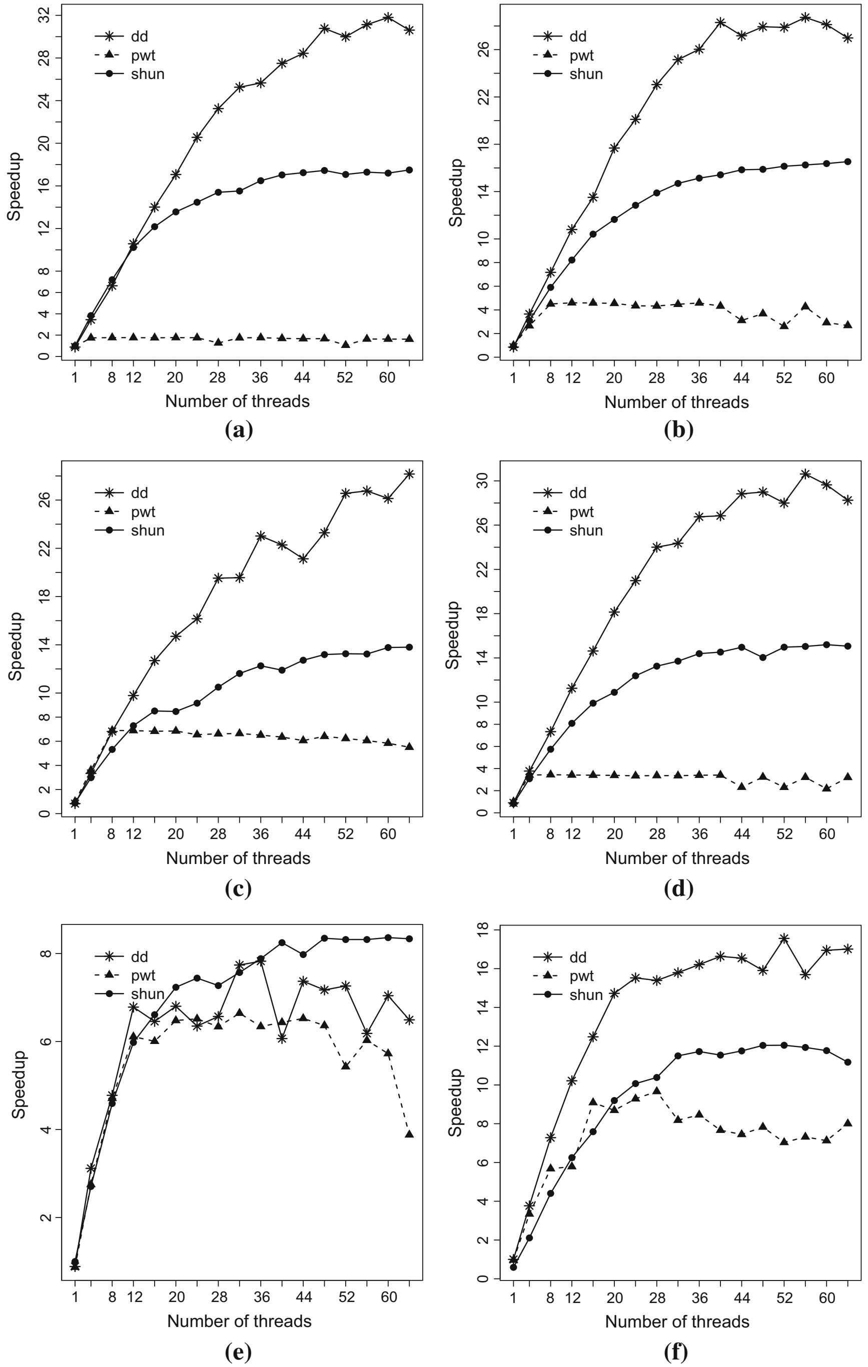


Fig. 5 Speedup with respect to the best sequential time. The caption of each figure indicates the name of the dataset, the input size n and the alphabet size σ. **a** *rna.3GB*, $n \approx 2^{32}$, $\sigma = 4$. **b** *prot*, $n \approx 2^{30}$, $\sigma = 27$. **c** *src. 200MB*, $n \approx 2^{28}$, $\sigma = 230$. **d** *en.4.30*, $n = 2^{30}$, $\sigma = 16$. **e** *src.2GB*, $n \approx 2^{29}$, $\sigma \approx 2^{21}$. **f** *en.14.30*, $n = 2^{30}$, $\sigma = 2^{14}$

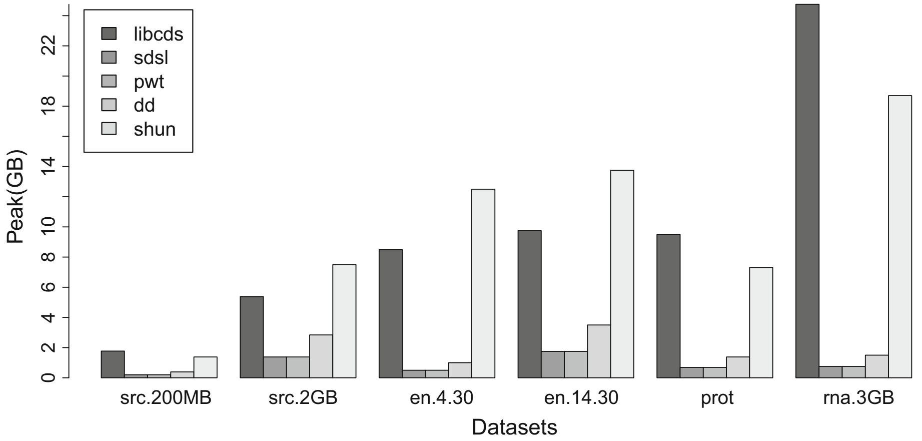


Fig. 6 Memory consumption sorted by n

of the *wtree*, such as number of bits, the range of the alphabet, and the offset. In our algorithms and in sdsl, memory consumption is dominated by the arrays which store offset values, not by the input sequence.

The main drawback of dd with respect to our own pwt is its memory consumption, since the latter increases with the alphabet size and the number of threads. For small alphabets, the working space of dd is almost constant. For instance, memory consumption for rna.2GB is 1 GB, plus a small overhead for each new thread. For larger alphabets, such as src.2GB with $\sigma \approx 2^{22}$, the working space increases linearly with the number of threads, using 1.46 GB with 1 thread and 2.5 GB with 12 threads. Fortunately, most of the sequences used in real-world applications have an alphabet size smaller than 2^{17}. Such is the case of DNA sequences, the human genome, natural language alphabets (Unicode standard), etc.[10]

4.4 Other experiments

In order to have a better understanding of our algorithms, we performed the following experiments:

Limited resources When memory is limited, algorithms such as LIBCDS and shun suffer a decrement in their performance. This is evident in Fig. 7, where we tested the parallel algorithms with datasets prot and src.1GB[11] on a 12-core computer with 6GB of DDR3 RAM.[12] In this new set of experiments, the speedup of our algorithms exceeded the speedup shown by shun, both for datasets where we previously showed the better performance (see Fig. 5b) and for datasets where previously shun showed better performance (see Fig. 5e; Table 2).

[10] The Unicode Consortium: http://www.unicode.org/.

[11] The construction times of shun with the src.2GB dataset exceeds 1 h. To make the algorithms in the figures comparable, we report the running times for the dataset src.1GB.

[12] The computer tested is a dual-processor Intel® Xeon® CPU (E5645) with six cores per processor, for a total of 12 physical cores running at 2.50GHz. Hyperthreading was disabled. The computer runs Linux 3.5.0-17-generic, in 64-bit mode. This machine has per-core L1 and L2 caches of sizes 32KB and 256KB, respectively, and 1 per-processor shared L3 cache of 12MB, with a 5,958MB (∼ 6GB) DDR3 RAM.

Fig. 7 Running experiments in a machine with limited resources

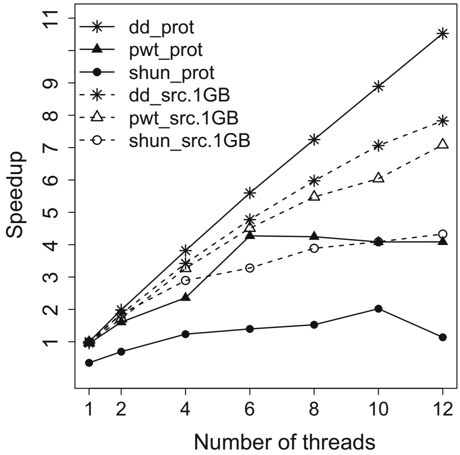
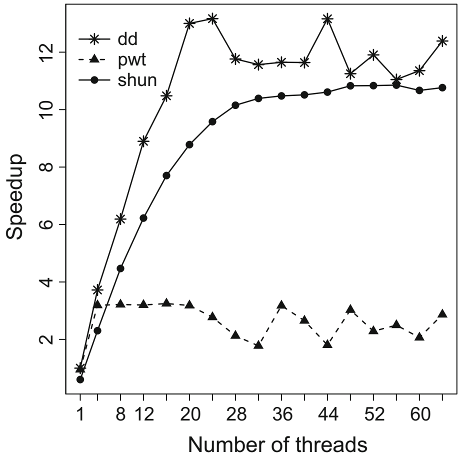


Fig. 8 Speedup of the dataset *en.4.30* encoding each symbol with 4 bytes



Encoding We observed that the encoding of the symbols of the original sequence has a great impact in the speedups of the construction algorithms. Figure 5a–d shows speedups greater than 27x, while there is a noticeable performance degradation in Fig. 5e, f. This is due to an encoding subtlety: The datasets used in the experiments resulting in Fig. 5a–d are encoded using one byte, while the other used four bytes. To prove the impact of the encoding in the performance of the construction algorithms, we repeated the experiments using a dataset that used four bytes per symbol for $\sigma \leq 2^8$. Figures 5d and 8 show the influence of encoding. As expected, the greater the memory used for encoding, the worse the performance. On multicore architectures, some levels of the memory hierarchy are shared by different cores. This increases the rate of memory evictions. Hence, it is crucial to reduce the number of memory transfers. Besides, in NUMA architectures, where each NUMA node has a local RAM and the transfers between local RAMs is expensive, the reduction in memory transfers is critical. In the case of one byte per symbol, each memory transfer carries four times more symbols than in the case of four bytes per symbol, effectively helping reduce memory transfers.

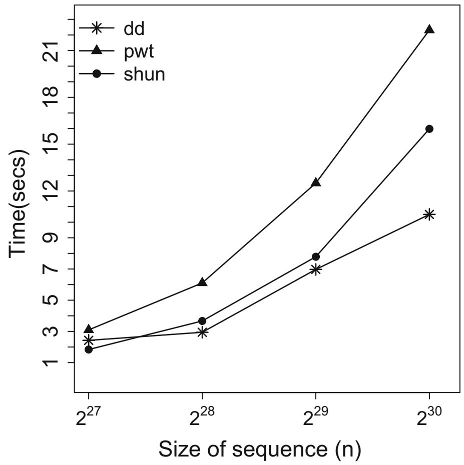


Fig. 9 Time over n with $\sigma = 2^{14}$, 64 threads and *en.14* datasets

Influence of the size of sequence Figure 9 shows that for the `en.14` dataset, fixing the number of threads to 64 and σ to 2^{14}, for larger n the domain decomposition algorithm behaves better in running time than the `pwt` algorithm and Shun's algorithm. In other words, with more cores and enough work for each parallel task, the `dd` algorithm should scale appropriately.

Influence of the locality of reference Theoretically, fixing n and varying σ with $p = \lg \sigma$ threads, the `pwt` algorithm should show a constant-time behavior, no matter the value of σ. However, in practice, the running times of `pwt` increase with the alphabet size. The reason for this difference in theoretical and practical results is that levels closer to the leaves in the *wtree* exhibit a weaker locality of reference. In other words, locality of reference of the `pwt` algorithm is inversely proportional to σ. Additionally, the dynamic multithreading model assumes that the cost of access to any position in the memory is constant, but that assumption is not true in a NUMA architecture. In order to visualize the impact of the locality of reference over running times, we generate two artificial datasets with $n = 2^{30}$, $\Sigma = \{1 \dots 2^y\}$, with $y \in \{4, 6, 8, 10, 12, 14\}$ and encoding each symbol with four bytes. The first dataset, `cont`, was created writing n/σ times each symbol of Σ and then sorting the symbols according to their position in the alphabet. The second dataset, `rand`, was created in a similar fashion, but writing symbols at random positions. The objective of the `cont` dataset is to force the best case of the `pwt` algorithm, where the locality of reference is higher. In contrast, the `rand` dataset forces the average case, with a low locality of reference. In these experiments, we used the optimal number of threads of `pwt`, that is, $p = \lg \sigma$. Besides, we allocated evenly the memory over the NUMA nodes to ensure constant access cost to any position in the memory.[13] The results are shown in Fig. 10. In its average case, dashed lines, the performance of the `pwt` algorithm is degraded for larger alphabets because locality of reference is low, increasing the amount of cache misses, and thus degrading the overall performance. In the best case, solid lines, the `pwt` shows a practical behavior similar to the theoretical one. Since the `dd` algorithm implements the `pwt` algorithm to build each partial *wtree*, the locality of reference impacts also on its performance. However, because the construction of the partial *wtree*s

[13] To ensure the constant access cost, we use the *numactl* command with "interleave = all" option. The command allocates the memory using round robin on the NUMA nodes.

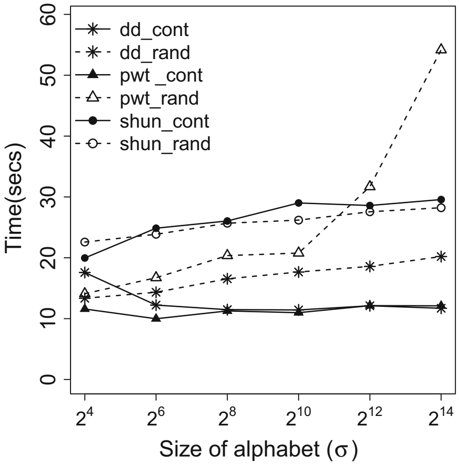


Fig. 10 Time over σ for the best and average cases with $n = 2^{30}$ and $p = \lg\sigma$ threads

involves sequences of size $O(n/p)$, the impact is less than in the pwt algorithm. Finally, Shun's algorithm is insensitive to the distribution of the symbols in the sequence.

The study of the impact of the architecture on the construction of *wtree*s and other succinct data structures, and the improvement in the locality of reference of our algorithms are interesting lines for future research.

4.5 Discussion

In most cases, the domain decomposition algorithm, dd, showed the best speedup. Additionally, dd can be adjusted in favor of either running time or memory consumption. pwt showed good scalability, but up to $p < \lg\sigma$. This limitation may be overcome by using pwt as part of dd, dividing the input sequence in an adequate number of subsequences.

With respect to working space, pwt was the algorithm with lowest memory consumption. This is important because an algorithm with low memory consumption can be executed in machines with limited resources, can reduce cache misses due to invalidations (*false sharing*) and can therefore reduce energy consumption. Even though memory consumption of the dd algorithm increases with the number of subsequences, it can be controlled manipulating the number of segments. In the case of shun, its memory consumption is too large to be competitive in machines with limited memory.

The encoding and the distribution of the symbols of the input sequence impact the performance of the algorithms. All the parallel algorithms introduced here show a better speedup for encodings that use less bits because there are less memory transfers. Our algorithms are also sensitive to the distribution of the symbols. When the symbols are randomly distributed, the locality of reference is worse in comparison with more uniform distributions. This gives us a hint to improve the performance of our algorithms in the future.

To sum up, in general, the dd algorithm is the best alternative for the construction of *wtree*s on multicore architectures, considering both running time and memory consumption. For domains with limited resources, pwt, which is a building block of dd, arises as a good alternative on its own.

5 Conclusions and future work

Despite the vast amount of research on wavelet trees, very little has been done to optimize them for current ubiquitous multicore architectures. We have shown that it is possible to have practical multicore implementations of wavelet tree construction by exploiting information related to the levels of the *wtree*, achieving $O(\lg n)$-time construction and good use of memory resources.

In this paper, we introduced two multicore algorithms for parallel construction of *wtree*s. Our domain decomposition algorithm, `dd`, may be used in any domain, but in those contexts where it is not possible to use all available resources, our per-level algorithm, `pwt`, may be more suitable. We have focused on the most general representation of a *wtree*, but some of our results may apply to other variants. For example, it would be interesting to study how to extend our results to compressed wavelet trees (e.g., Huffman-shaped *wtree*s) and to generalized wavelet trees (i.e., multiary wavelet trees where the fan out of each node is increased from 2 to $O(polylog(n))$). Also, it is interesting to explore the extension of our results to the Wavelet Matrix [8] (a different level-wise approach to avoid the $O(\sigma \lg n)$ space overhead for the structure of the tree, which turns out to be simpler and faster than the wavelet tree without pointers). Future work also involves dynamization, whereby the *wtree* is being modified concurrently by many processes as it is queried, through dynamic succinct data structures, even sequential ones is still an open area of research. A further line of work involves the design of cache-aware algorithms to construct *wtree*s, obtaining more efficient implementations, both in time and in memory resources. In our previous work [15], we studied the parallelization of some queries on *wtree*s. The parallelization of other queries is yet another interesting future work.

For all our construction algorithms, we assume that the input sequence S fits in memory. However, we can extend our results to the construction of *wtree*s where the input sequence S and the *wtree* do not fit. Following some implementation ideas of SDSL [16], we can read the input sequence in buffers to construct partial *wtree*s for each buffer and finally merge all of them to obtain the final *wtree*. In more detail, we can extend our algorithms as follows:

1. Read the input sequence S using a buffer of size b. We can use the portion of main memory that will not be used by the *wtree* as the buffer.
2. Create a partial *wtree* without rank/select structures taking the buffer as input. The partial *wtree* can be constructed in parallel using our `dd` algorithm with $O(b \lg \sigma / p)$ time and $O(1)$ span. (We could also use the `pwt` if the available memory is scarce.) The starting position of each node in the partial *wtree* is stored in a bidimensional array L.
3. Repeat steps 1 and 2 until the complete input sequence is read.
4. After the complete input sequence is read, we compute the final position of the nodes of all the partial *wtree*s. These positions are computed performing a parallel prefix sum[19] over the values of the arrays L's, similar to the `dd` algorithm. It takes $O(b\sigma/p + \lg p)$ time and $O(\lg(b\sigma))$ span.
5. The final *wtree* is constructed using Function mergeBA with $O(n \lg \sigma / pw)$ and $O(1)$ span, where w is the word size of the architecture.

The extension takes $O(n \lg \sigma / p + b\sigma/p + \lg p)$ time and $O(n/b + \lg(b\sigma))$ span. Notice that this idea is similar to the `dd` algorithm, and it can be applied on multiple levels. For example, it can be used on distributed architectures, where the buffers are processed by different machines, and one machine merges all the partial *wtree*s. Additionally, observe that we can use the entire main memory as the buffer, storing the partial *wtree*s and the L arrays on disk each time we finish the processing of a buffer. We leave the implementation and empirical evaluation of these ideas as future work.

It has become evident that architecture has become relevant again. It is nowadays difficult to find single-core computers. Therefore, it seems like a waste of resources to stick to sequential algorithms. We believe one natural way to improve performance of important data structures, such as wavelet trees, is to squeeze every drop of parallelism of modern multicore machines.

Acknowledgements This work was supported in part by the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie Grant Agreement No. 690941 and the doctoral scholarships of CONICYT Nos. 21120974 and 63130228 (first and second authors, respectively). We also would like to thank Roberto Asín for making his multicore computers, Mastropiero and Günther Frager, available to us.

References

1. Arroyuelo D, Costa VG, González S, Marín M, Oyarzún M (2012) Distributed search based on self-indexed compressed text. Inf Process Manag 48(5):819–827. doi:10.1016/j.ipm.2011.01.008
2. Bingmann T (2015) malloc_count—tools for runtime memory usage analysis and profiling. http://panthema.net/2013/malloc_count/ (2013). Last accessed: 17 Jan 2015
3. Blumofe RD, Leiserson CE (1999) Scheduling multithreaded computations by work stealing. J ACM 46(5):720–748. doi:10.1145/324133.324234
4. Brisaboa NR, Luaces MR, Navarro G, Seco D (2013) Space-efficient representations of rectangle datasets supporting orthogonal range querying. Inf Syst 38(5):635–655. doi:10.1016/j.is.2013.01.005
5. Burrows M, Wheeler DJ (1994) A block-sorting lossless data compression algorithm. Tech. rep., Digital Equipment Corporation
6. Claude F (2011) A compressed data structure library. https://github.com/fclaude/libcds. Last accessed: 13 August 2015
7. Claude F, Navarro G (2009) Practical rank/select queries over arbitrary sequences. In: SPIRE. Springer, Berlin, pp 176–187. doi:10.1007/978-3-540-89097-3_18
8. Claude F, Navarro G (2012) The wavelet matrix. In: SPIRE, vol 7608. Springer, Berlin, pp 167–179. doi:10.1007/978-3-642-34109-0_18
9. Claude F, Nicholson PK, Seco D (2011) Space efficient wavelet tree construction. In: SPIRE, vol 7024. Springer, Berlin, pp 185–196
10. Cormen TH, Leiserson CE, Rivest RL, Stein C (2009) Introduction to algorithms, 3rd edn., chap. Multithreaded algorithms. The MIT Press, Cambridge, pp 772–812
11. Faro S, Külekci MO (2012) Fast multiple string matching using streaming SIMD extensions technology. In: SPIRE. Springer, Berlin, pp 217–228. doi:10.1007/978-3-642-34109-0_23
12. Ferragina P, Manzini G (2000) Opportunistic data structures with applications. In: Proceedings of the 41st annual symposium on foundations of computer science, FOCS '00. IEEE Computer Society, Washington, DC, USA, p 390. http://dl.acm.org/citation.cfm?id=795666.796543
13. Ferragina P, Manzini G, Mäkinen V, Navarro G (2004) String processing and information retrieval: 11th international conference, SPIRE 2004, Padova, Italy, 5–8 October 2004. Proceedings, chap. An Alphabet-Friendly FM-Index. Springer, Berlin, pp 150–160. doi:10.1007/978-3-540-30213-1_23
14. Ferragina P, Manzini G, Mäkinen V, Navarro G (2007) Compressed representations of sequences and full-text indexes. ACM Trans Algorithms 3(2):20. doi:10.1145/1240233.1240243
15. Fuentes-Sepúlveda J, Elejalde E, Ferres L, Seco D (2014) Efficient wavelet tree construction and querying for multicore architectures. In: Gudmundsson J, Katajainen J (eds) Experimental algorithms, Lecture Notes in Computer Science, vol 8504. Springer, Berlin, pp 150–161. doi:10.1007/978-3-319-07959-2_13
16. Gog S (2015) Succinct data structure library 2.0. https://github.com/simongog/sdsl-lite (2012). Last accessed: 17 Jan 2015
17. González R, Grabowski S, Mäkinen V, Navarro G (2005) Practical implementation of rank and select queries. In: WEA. CTI Press, Greece, pp 27–38. Poster
18. Grossi R, Gupta A, Vitter JS (2003) High-order entropy-compressed text indexes. In: SODA. Soc. Ind. Appl. Math., Philadelphia, pp 841–850
19. Helman DR, JáJá J (2001) Prefix computations on symmetric multiprocessors. J Parallel Distrib Comput 61(2):265–278. doi:10.1006/jpdc.2000.1678

20. Illumina, Inc. (2016) An introduction to next-generation sequencing technology. http://www.illumina.com/content/dam/illumina-marketing/documents/products/illumina_sequencing_introduction.pdf
21. Ladra S, Pedreira O, Duato J, Brisaboa NR (2012) Exploiting SIMD instructions in current processors to improve classical string algorithms. In: ADBIS. Springer, Berlin, pp 254–267. doi:10.1007/978-3-642-33074-2_19
22. Makris C (2012) Wavelet trees: a survey. Comput Sci Inf Syst 9(2):585–625
23. Matsumoto M, Nishimura T (1998) Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. ACM Trans Model Comput Simul 8(1):3–30. doi:10.1145/272991.272995
24. Navarro G (2012) Wavelet trees for all. In: CPM. Springer, Berlin, pp 2–26. doi:10.1007/978-3-642-31265-6_2
25. Navarro G, Nekrich Y, Russo LMS (2013) Space-efficient data-analysis queries on grids. Theor Comput Sci 482:60–72. doi:10.1016/j.tcs.2012.11.031
26. Pantaleoni J, Subtil N (2016) Nvbio library. http://nvlabs.github.io/nvbio/index.html. Accessed 12 April 2016
27. Raman R, Raman V, Satti SR (2007) Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. ACM Trans Algorithms 3(4):43. doi:10.1145/1290672.1290680
28. Schnattinger T, Ohlebusch E, Gog S (2012) Bidirectional search in a string with wavelet trees and bidirectional matching statistics. Inf Comput 213:13–22. doi:10.1016/j.ic.2011.03.007. http://www.sciencedirect.com/science/article/pii/S0890540112000235. Special Issue: Combinatorial Pattern Matching (CPM 2010)
29. Shun J (2015) Parallel wavelet tree construction. In: Proceedings of the IEEE data compression conference, Utah, USA, pp 63–72. doi:10.1109/DCC.2015.7
30. Singer J (2012) A wavelet tree based fm-index for biological sequences in SeqAn. Master's thesis, Freie Universität Berlin. http://www.mi.fu-berlin.de/wiki/pub/ABI/FMIndexThesis/FMIndex.pdf
31. Tischler G (2011) On wavelet tree construction. In: CPM. Springer, Berlin, pp 208–218
32. Touati SAA, Worms J, Briais S (2013) The Speedup-Test: a statistical methodology for program speedup analysis and computation. Concurr Comput Pract Exp 25(10):1410–1426. doi:10.1002/cpe.2939. https://hal.inria.fr/hal-00764454. Article first published online: 15 Oct 2012
33. Välimäki N, Mäkinen V (2007) Space-efficient algorithms for document retrieval. In: CPM, LNCS, vol. 4580. Springer, Berlin, pp 205–215. doi:10.1007/978-3-540-73437-6_22
34. Wetterstrand KA (2016) DNA sequencing costs: data from the NHGRI genome sequencing program (GSP). http://www.genome.gov/sequencingcosts. Accessed 12 April 2016

José Fuentes-Sepúlveda received his degree of Doctor in Computer Science from the Universidad de Concepción (Chile) in 2016. He is currently a postdoctoral researcher at the Universidad de Chile (Chile) under the supervision of Gonzalo Navarro. His research interests include multicore algorithms, construction of space-efficient data structures and practical parallel models.

Erick Elejalde received his M.S. degree in Computer Science from the Universidad de Concepción (Chile) in 2013. He is currently a Ph.D. student at the Department of Informatics Engineering and Computer Science, Faculty of Engineering, University of Concepción (Chile). His research interests include concurrent data structures, multicore algorithms, and big data analysis.

Leo Ferres received his Ph.D. from Carleton University in Ottawa, Canada. He is a Research Professor in the Data Science Institute, Faculty of Engineering at Universidad del Desarrollo in Santiago de Chile. He is also a Fellow of the Telefónica Research & Development Group. His interest include high-performance computing, multicore algorithms, and handling big, heterogeneous and mostly textual data. He collects old 1980s computers.

Diego Seco received his M.S. degree in Computer Science from the University of A Coruña (Spain) in 2006 and a Ph.D. in Computer Science from the same university in 2009. His research interests include algorithms and data structures, geographic information retrieval, geographic information systems, and space-efficient data structures for textual and geographic data. Nowadays, he is an associate professor at the Department of Informatics Engineering and Computer Science, Faculty of Engineering, University of Concepción (Chile).