Efficient Wavelet Tree Construction and Querying for Multicore Architectures*

José Fuentes-Sepúlveda, Erick Elejalde, Leo Ferres, Diego Seco

Universidad de Concepción, Concepción, Chile
{jfuentess,eelejalde,lferres,dseco}@udec.cl

Abstract. Wavelet trees have become very useful to handle large data sequences efficiently. By the same token, in the last decade, multicore architectures have become ubiquitous, and parallelism in general has become extremely important in order to gain performance. This paper introduces two practical multicore algorithms for wavelet tree construction that run in $O(n)$ time using $\lg \sigma$ processors, where n is the size of the input and σ the alphabet size. Both algorithms have efficient memory consumption. We also present a querying technique based on batch processing that improves on simple domain-decomposition techniques.

1 Introduction and motivation

After their introduction in the mid-2000s, *multicore* computers —computers with a shared main memory and more than one processing unit— have become pervasive. In fact, it is hard nowadays to find a single-processor desktop, let alone a high-end server. The argument for multicore systems is simple [20, 22]: thermodynamic and material considerations prevent chip manufacturers from increasing clock frequencies beyond 4GHz. Since 2005, clock frequencies have stagnated at around 3.75GHz for commodity computers, and even in 2013, 4GHz computers are rare. Thus, one possible next step in performance is to take advantage of multicore computers. To do this, algorithms and data structures will have to be modified to make them behave well in parallel architectures.

In the past few years, much has been written about compressed data structures. One such structure that has benefited from thorough research is the *wavelet tree* [12], henceforth *wtree*. Although the *wtree* was original devised as a data structure for encoding a reordering of the elements of a sequence[12, 9], it has been successfully used in many applications. For example, it has been used to index documents [24], grids [18] and even sets of rectangles [3], to name a few applications (w.r.t. [19, 16] for comprehensive surveys).

Our contributions in this paper are as follows: we first propose two linear $O(n)$ time parallel algorithms for the most expensive operation on wavelet trees, construction, using $\lg \sigma$ processors[1] (see Sect. 3). We report experiments which show the algorithms to be practical for large datasets, achieving close to perfectly

* This work was supported in part by CONICYT FONDECYT 11130377.

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

linear speedup (Sect. 4). In order to exploit multicore architectures, we also investigated techniques to speed up range queries and propose BQA (*batch-query-answering*), a hybrid domain-decomposition/parallel batch processing technique (see Sect. 3) that exploits both multicore architectures and cache data locality effects in hierarchical memory systems. We empirically achieve almost perfect linear throughput by augmenting the number of cores (see Sect. 4). To the best of our knowledge, this is the first proposal of a parallel wavelet tree (in the dynamic multithreading model, see Sect. 2).

2 Preliminaries

Wavelet trees: For the purpose of this paper, a wavelet tree 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. Wavelet trees can be stored in space bounded by different measures of the entropy of the underlying data, thus enabling compression. In addition, they can be implemented efficiently [5] and perform well in practice.

The wavelet tree 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.

In its simplest form, this structure 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, and supports aforementioned queries in $O(\lg \sigma)$ time by traversing the tree using $O(1)$-time *rank*/*select* operations on bitmaps [21]. Its construction takes $O(n \lg \sigma)$ time (we do not consider space-efficient construction algorithms [6, 23]). 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 [15], which is important for large alphabets. We focus on the simple form, though our results can be extended to other encodings and tree shapes.

The *wtree*s support more complex queries than the primitives described above. For example, Mäkinen and Navarro [15] showed its connection with a classical two-dimensional range-search data structure. They showed how to solve range queries in a wavelet tree and its applications in *position-restricted searching*. In [13], the authors represent posting lists in a *wtree* and solve ranked AND queries by solving several range queries synchronously. Thus, solving range queries *in parallel* becomes important. As we present a parallel version of these queries, let us define them here. Given $1 \leq i \leq i' \leq n$ and $1 \leq j \leq j' \leq \sigma$, a

range query $rq(S, i, i', j, j')$ reports all the symbols s_x such that $x \in [i, i']$ and $s_x \in [j, j']$. The counting version of the problem can be defined analogously.

Some work has been done in parallel processing of wavelet trees. In [1], the authors explore the use of wavelet trees in distributed web search engines. They assume a distributed memory model and propose partition techniques to balance the workload of processing wavelet trees. Note that our work is complementary to theirs, as each node in their distributed system can be assumed a multicore computer that can benefit from our algorithms. In [14], the authors explore the use of SIMD instructions to improve the performance of wavelet trees (and other string algorithms, see, for example, [8]). This set of instructions can be considered as low-level parallelism. We can also benefit from their work as it may improve the performance of the sequential parts of our algorithms.

Dynamic multithreading model: *Dynamic multithreading* (DYM) [7, Chapter 27] is a model of parallel computation which is faithful to several industry standards such as Intel's CilkPlus (`cilkplus.org`) and OpenMP Tasks (`openmp.org/wp`), and Threading Building Blocks (`threadingbuildingblocks.org`).

We will define a *multithreaded computation* 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 instructions; whereby in this case, u must be executed before v.[2] In order to signal parallel execution, we will 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 will group these instruction into a single *strand*. Strands are scheduled onto processors 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 [2].

To measure the efficiency of our parallel wavelet tree algorithms, we will use three metrics: the *work*, the *span* and the number of processors. In accordance to the parallel literature, we will subscript running times by P, so T_P is the running time of an algorithm on P processors. The *work* is the total running time taken by all (unit-time) strands when executing on a *single* processor (i.e., T_1)[3], while the *span*, denoted as T_∞, is the *critical path* (the longest path) of G. In this paper, we are interested in speeding up wavelet tree manipulation and finding out the upper 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. We

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

[3] Notice, again, that analyzing the work amounts to finding the running time of the serial algorithm using the RAM model.

also define *parallelism* as the ratio T_1/T_∞, the maximum theoretical speedup that can be achieved on *any* number of processors.

3 Multicore wavelet tree

Parallel construction: We focus on binary wavelet trees where the symbols of Σ are contiguous in $[1, \sigma]$. If they are not contiguous, a bitmap is used to remap the sequence to a contiguous alphabet [5]. Under these restrictions, the *wtree* is a balanced binary tree with $\lceil \lg \sigma \rceil$ levels.

In this scenario, a simple recursive algorithm, such as the one implemented in LIBCDS (`http://libcds.recoded.cl`), can build a *wtree* in $T_1 = O(n \lg \sigma)$ time by a linear processing of the symbols at each node. This algorithm works by halving Σ recursively into binary sub-trees whose left-child are all 0s and the right-child are all 1s, until 1s and 0s mean only one symbol in Σ. Instead, we propose an iterative construction algorithm that performs worse when executed sequentially, but shows nice parallel behavior. The key idea of the algorithm is that we can build any level of the *wtree* independently from the others. Unlike the classical construction, when building a level we cannot assume that a previous step is providing us the correct permutation of the elements of S. Instead, we compute the node at level i for each symbol of the original sequence. The following proposition shows how it can be computed.

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 gives us its corresponding node at level i. In the word-RAM model with word size $\Omega(\lg n)$, this computation takes $O(1)$ time, and thus 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.*

The iterative parallel construction procedure is shown in Algorithm 1 (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). The output is a *wtree* WT, which represents the input data S. We denote the ith level of WT as WT_i, $\forall i, 0 \leq i < \lceil \lg \sigma \rceil$ and the jth node in level WT_i as WT_i^j, $\forall j, 0 \leq j < 2^i$.

The outer loop (line 3) iterates in parallel over $\lceil \lg \sigma \rceil$ levels. Lines 4 to 13 scan each level performing the following tasks: the first step (lines 4 to 7) calculates the maximum number of nodes for the current level, and traverse it initializing each node and its bitmap. For this initialization we can compute the size of each node with a linear time sweep of the elements in the node. The second step (lines 8 to 13) computes for each symbol in S, the node that represents the alphabet range that holds it at the current level (line 9 show an equivalent representation of the idea in Proposition 1). Then, the algorithm computes whether the symbol belongs to either the first or second half of the part of Σ that represents that

Input : S, n, σ
Output: A wavelet tree representation WT of S

```
1  l ← ⌈lg σ⌉
2  WT ← Create a new tree with l levels
3  parfor i ← 0 to l − 1 do
4  |  m ← 2^i
5  |  WT_i ← Create a new level with m nodes
6  |  parfor j ← 1 to m do
7  |  |  WT_i^j ← Initialize a new Node
8  |  for v ← 1 to n do
9  |  |  u ← ⌊s_v/⌊σ/(2×m)⌋⌋
10 |  |  if u is odd then
11 |  |  |  bitmapSetNextBit(WT_i^{u/2}.bitmap, 1)
12 |  |  else
13 |  |  |  bitmapSetNextBit(WT_i^{u/2}.bitmap, 0)
14 return WT
```

Algorithm 1: Wavelet tree parallel construction (`pwt`)

node. Notice that *bitmapSetNextBit* needs to keep track of the positions already written in the bitmap and set the value of the next bit. When we reach the last element, all the bitmaps contain the necessary information for that level. Finally, the bitmaps in this structure need to support rank/select operations, thus the construction algorithm must create additional structures after the bitmaps are completed. Notice that the **parfor** starting at line 6 scans the nodes of level i initializing them. The number of nodes grows exponentially larger when more levels are created, until we reach n nodes. This brings about a task workload imbalance among the worker threads because any given task may have exponentially more work to do. To prevent this, we also divide the first step into strands that the work-stealing scheduler will balance (see Sect. 2). It is easy to see that a sequential version of this algorithm takes $O(n \lg \sigma)$ time, which matches the time for construction found in the literature for non space-efficient construction algorithms. If **parfor** implements parallelism in a "divide-and-conquer" fashion (as in our model and implementation), then the DAG represents a binary-tree of constant-time division of Σ until it reaches the leaves of said tree, each of which has $O(n)$ weight. The work of `pwt` is still $T_1 = O(n \lg \sigma)$. All paths in the DAG, however, are the same length, and the same weight: the internal nodes are all $O(1)$, and the leaves are all $O(n)$. Thus, the critical path is $T_\infty = O(n)$ in all cases. In the same vein, parallelism will be $T_1/T_\infty = O(\lg \sigma)$, again for all cases. It follows that having $P = \lg \sigma$ will be enough to obtain optimal speedup.

The main drawback of the `pwt` algorithm is that it only scales 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 up to $\lg \sigma$ processors. In order to make an efficient use of all available cores we followed a different approach. This time we split the input data instead of the steps of the algorithm. In this way, the new algorithm creates as many chunks of the text as available cores, allowing

it to take full advantage of the resources at hand. The algorithm then executes the construction for each substring and finally merges all resulting *wtree*s into a single one that represents the entire input text. We called this algorithm `dd` because of its domain decomposition nature. Although the `dd` algorithm is easy to follow, the merging part is not trivial. Here the order in which the partial trees are merged is important. In the merge step, the algorithm assigns an entire level i to each thread. Each thread concatenates the corresponding bitarrays of each tree, creating the bitarray for the final tree in the assigned level i. The concatenation takes the range of bits corresponding to each WT_i^j node of the level, in each tree and the result is the range of bits associated to the WT_i^j node in the final *wtree*. Concatenation is done following the same relative order of the substrings with respect to the input text. The procedure is repeated for each node in the level. In this case the thread working with the last level (the leaves) will do most of the work. Although all levels have the same size in bits, the last level is the one with the greatest number of calls to the concatenation function (one for each virtual node), leading to a bigger overhead.

The `dd` algorithm has the same asymptotic complexity for the total work $T_1 = O(n \lg \sigma)$. When running on P processors the construction of the partial trees takes $O(\frac{n}{P} \log \sigma)$ time. Merge takes $O(\frac{n \log \sigma}{PW})$, where W is the size of the computer word in that architecture. Note that merge has a linear speedup while $P \leq \log \sigma$. Thus, if we consider W as a constant, the *span* is again $T_\infty = O(n)$ in all cases.

Parallel querying: We distinguish between two kinds of queries on wavelet trees: *path* and *branch* queries. Path queries are characterized by following just a single path from the root to a leaf and the value in level $i-1$ has to be computed before the value in level i. Examples of this type of queries are *select*, *rank*, and *access*. On the other hand, branch queries may follow more than one path root-to-leaf (indeed they may reach more than one leaf). Each path has the same characteristics as path queries and each path is independent from others paths. Examples of this type of queries are *range count* and *range report* [10].

In a parallel setting, a single path query cannot be parallelized because only one level of the query can be computed at a time. The common alternative is parallelizing several path queries using domain decomposition over queries (i.e., dividing queries over P). For this naïve approach, we obtained near-optimal throughput, defined as the number of processors times sequential throughput. We do not report this experiment for lack of space.

We implemented two branch-query-answering techniques: *individual*-query-answering (IQA) and *batch*-query-answering (BQA). The IQA technique is the obvious query by query processing. The BQA technique involves grouping sets of queries to take advantage of spatial and temporal locality in hierarchical memory architectures. For instance, at each node in the *wtree*, we can evaluate all the queries in a batch reusing the node's bitarray, thus increasing locality.

With little effort, we can parallelize sequential IQA in a domain decomposition fashion (denoted as `dd-IQA`), achieving near-optimal throughput (more than nine times the throughput for $P = 12$ compared to the sequential IQA).

Input : WT_i^j, *batch*, *num_queries*, *states*, *results*
Output: *results*: A collection containing the results for each query

```
1  lbatch ← a new collection with num_queries elements
2  rbatch ← batch
3  local_states ← a new collection with num_queries elements
4  for q ← 1 to num_queries do
5  |  local_states_q ← states_q
6  |  if local_states_q = 1 then continue
7  |
8  |  if batch_q.x^s > batch_q.x^e ∨ (isLeaf(WT_i^j) ∩ batch_q.y_range) = ∅ then
9  |  |  local_states_q ← 1
10 |  |  continue
11 |  if isLeaf(WT_i^j) then
12 |  |  results_q^j ← batch_q.x^e − batch_q.x^s + 1      /* j is the label */
13 |  |  continue
14 |  lbatch_q.x^s ← rank_0(WT_i^j.bitmap, batch_q.x^s − 1) + 1
15 |  lbatch_q.x^e ← rank_0(WT_i^j.bitmap, batch_q.x^e)
16 |  rbatch_q.x^s ← rank_1(WT_i^j.bitmap, batch_q.x^s − 1) + 1
17 |  rbatch_q.x^e ← rank_1(WT_i^j.bitmap, batch_q.x^e − 1)
18 |  lbatch_q.y_range ← batch_q.y_range
19 if ∀_q, local_states_q = 1 then return
20
21 spawn parallelBQA(WT_{i+1}^{2j}, lbatch, num_queries, local_states, results)
22 parallelBQA(WT_{i+1}^{2j+1}, rbatch, num_queries, local_states, results)
23 return                                  /* implicit sync */
```

Algorithm 2: Parallel batch querying of range report (`parallelBQA`)

The `parallelBQA` algorithm is shown in Algorithm 2. It implements the general BQA technique mentioned above, answering queries in a single batch. In particular, the algorithm portrayed here parallelizes the recursive calls during the traversal of the *wtree* (the **spawn** instruction in line 19). This is what we call "internal" parallelization. If the **spawn** is taken out, we would be left with a sequential batch processing algorithm. In addition, if P is sufficiently large, we can also apply domain decomposition techniques to the list of batches, calling `parallelBQA` also in parallel (denoted here as `dd-parallelBQA`). This means a "double" parallelization, an internal one and an external one. Notice that although the algorithm implements *range report* this technique is also applicable to *range count*.

4 Experimental results

All algorithms were implemented in the C programming language and compiled using GCC 4.8 using the -O3 optimization flag. The experiments were carried out on a dual-processor Intel Xeon CPU (E5645) with six cores per processor,

for a total of 12 physical cores running at 2.40GHz. 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 a per-processor shared L3 cache of 12MB, with a 5,958MB (∼6GB) DDR RAM memory. Algorithms were compared in terms of running times using the usual high-resolution (nanosecond) C functions in `<time.h>`[4].

Construction experiments: We compared the implementation of our parallel wavelet tree construction algorithms, considering one pointer per node and one pointer per level, against the best current implementations available: LIBCDS and SDSL[5]. Both libraries were compiled with their default options. In particular, LIBCDS was compiled with optimization -O9 and SDSL with optimization -O3. Regarding to the bitarray implementation, in our implementations and in LIBCDS we use the 5%-extra space structure presented in [11]. In SDSL we use its `bit_vector_il` implementation, which yielded the best time performance in our experiments. The "trials" consisted in manipulating different alphabet and input sizes (i.e., σ and n), and the number of processors P recruited to work on the task. Since we are interested in big data, in order to obtain large enough alphabets, we took the `english` corpus of the Pizza & Chili website[6] as a sequence of *words* instead of the characters of the English alphabet. This representation gave us a large initial alphabet Σ of σ=633,816 symbols, which was ordered by their frequency of occurrence in the original `english` text. For experimentation, we generated an alphabet Σ' of size 2^k, taking the top 2^k *most frequent* words in the original Σ, and then assigning to each symbol a random index in the alphabet using a Marsenne Twister [17], with $k \in \{4, 6, 8, 10, 12, 14\}$. To create the input sequence S of n symbols, 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' (∼1.5GB, in the case of $\sigma' = 2^{14}$), maintaining the order of the original `english` text. We then either split S until we reached the target sizes, which varied from 2MB (i.e., 2^{21} bytes), 4, 8, 16, 32, 64, 128, 256 up to 512MB (the maximum in the `english` corpus), or concatenated S with initial sub-sequences of itself to reach larger sizes up to 2GB (2^{31} bytes)[7].

We repeated each trial three times. We worked with the minimum time taken by all three repetitions of a trial, assuming that slightly larger values for any given trial are just "noise" from external processes such as operating system and networking tasks. In our experiments, LIBCDS showed better times than SDSL, hence we consider LIBCDS as the baseline to report speedup.

Fig. 1 shows the speedup varying the number of threads. The figure shows that algorithms scale with the number of threads, up to 12 threads, because the

[4] This is a reproducible-research-friendly paper, everything needed to replicate these results is available at `www.inf.udec.cl/~leo/sea2014`

[5] `https://github.com/simongog/sdsl-lite`. We thank Simon Gog for the conversations about the use and implementation of SDSL.

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

[7] Notice that since we worked with integers, these numbers expressed in bytes should be divided by 4 (in our architecture) to get the number of elements n of S.

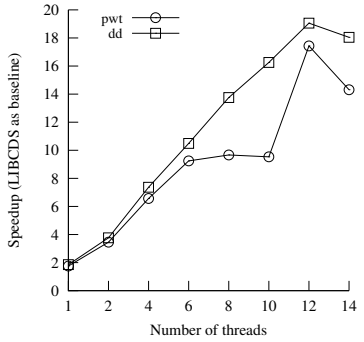

Fig. 1: Speedup over threads with a 2GB text and $\sigma = 2^{12}$.

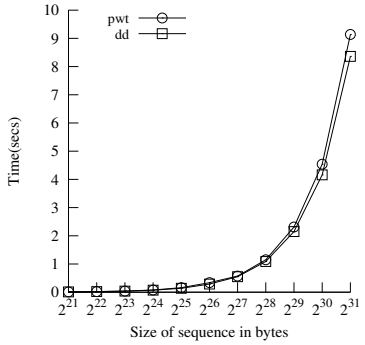

Fig. 2: Time over n, with $\sigma = 2^{12}$ and 12 threads.

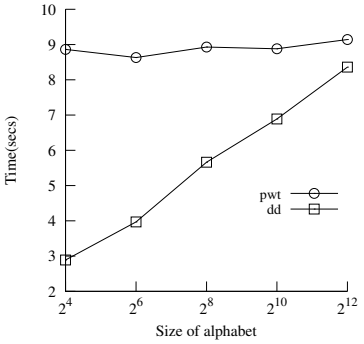

Fig. 3: Time over σ with a 2GB text and 12 threads.

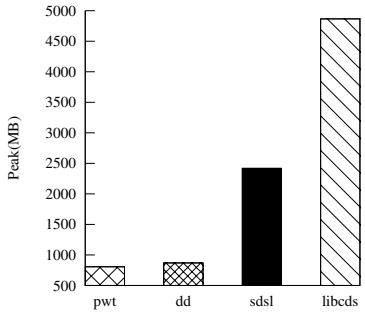

Fig. 4: Memory consumption with a 2GB text and $\sigma = 2^{12}$.

hardware has 12 processors. With more processors and enough work for each thread, our algorithms should scale appropriately. Upon reaching 12 threads, the operating system has to schedule the excess tasks on the same processors, slowing down the general construction process. The graph shows a maximum speedup greater than the number of cores. However, this should not be considered a *super-linear speedup* (i.e., a speedup $> P$). Speedups shown in Fig. 1 were calculated using LIBCDS, the best current sequential implementation. It is necessary to consider that LIBCDS as a library may have an extra overhead in its performance, but this will affect it up to a constant factor.

The time of both algorithms is dominated by the thread that has the most work to do. In the case of `dd`, all threads have the same amount of work and curve scales linearly. However, `pwt` has a different behaviour. Fig. 1 shows the construction of a wavelet tree with 12 levels, the time will be dominated by the thread that will build more levels. In the case of six threads, each thread has to build two levels. In the case of 8 and 10 threads, four and two threads, respectively, will have to build two levels, with a time similar to the case of 6 threads. Finally, with 12 threads, each thread has to build only one level.

Fixing the number of threads to 12 and varying n and σ, we obtain the results of Fig(s). 2 and 3, respectively. Fig. 2 shows that `dd` has the best performance, followed closely by `pwt`. Fig. 3 shows an interesting result. When varying the size of the alphabet, the times of `pwt` are almost constant whereas the times of `dd` increase. This shows that in domains where σ is small, `dd` is the best approach, mainly because it has a better behaviour in memory transfers, yielding less cache misses. However, for larger σ, which results in *wtree* structures with more levels, the `pwt` shows better scalability. This is useful, for example, for gene encoding, grids, natural language vocabularies, etc.

We also report memory consumption in Fig. 4. To measure it, we monitored the memory allocated with *malloc* and released with *free*, taking the peak of consumption. We only considered the memory allocated during the construction, but not the memory allocated to store the text. The `pwt` and `dd` implementations do not need to copy the input sequence as Libcds and Sdsl. Instead, they just read the input sequence. In particular, the memory consumption of `dd` depends on P, n and σ: $O(P\sigma \lg(n))$, because each thread maintains the topology of the tree to do the merge later. The memory consumption of `pwt` depends on n and σ, but does not depend on P. In all algorithms tested, we assumed that the input sequence cannot be destroyed. If they could, Sdsl and Libcds would save space. By the same token, `dd` and `pwt` algorithms could be even faster if they were allowed to use more memory resources like Libcds and Sdsl do. Given the improvement in performance we achieve using parallelism, we explicitly designed our algorithms to save memory and still achieve significant speedups.

Querying experiments: To generate *branch queries*, ranges over the text were selected with random bounds and the size was fixed at 1%. In order to stress the querying algorithm, we also took $[1, \sigma]$ as the range of the alphabet. This ensured that the query traversal reached the leaves of the *wtree*. To compare sequential IQA and parallel IQA, we randomly generated 10,000 range queries. To test the BQA techniques we took a new set of randomly generated 10,000 queries and grouped them into 100-query batches. As we did for construction experiments, all *branch query* experiments were tested on the 2GB `english` text, $\sigma' = 2^{14}$ and varying the available processors from 1 to 12 (see Fig. 5). We repeated each experiment three times. We worked with the maximum throughput taken by all three repetitions, similar to Sect. 4.

As discussed in Sect. 3, the BQA technique implies a little more programming effort but improves throughput over the IQA by answering 10% more queries/second in the sequential case and over 23% more queries/second for the domain-decomposition case (denoted here as `dd-BQA`). As we saw, a consequence of the BQA is that tasks now demand enough work to offset the cost of *internally* parallelizing the batch answering process (see Algorithm 2). By combining domain decomposition with internal batch parallelization, we achieve 34% more throughput (i.e., we answer one third more queries a second) compared to the domain-decompositioned IQA. Throughput scales well over P. Table 1 shows the running times of the different implementations.

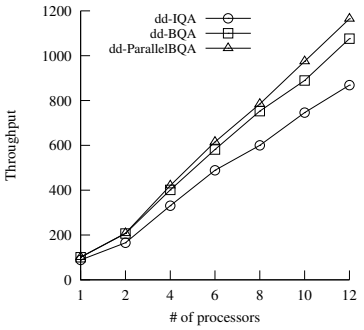

Fig. 5: Throughput over P for 100 batches of 100 queries (10,000 queries).

	Threads						
Algorithms	**1**	**2**	**4**	**6**	**8**	**10**	**12**
IQA	109.2	-	-	-	-	-	-
dd-IQA	113	60.6	30.3	20.5	16.7	13.4	11.5
BQA	99.2	-	-	-	-	-	-
dd-BQA	99.1	48.4	24.9	17.2	13.3	11.2	9.3
parallelBQA	98.5	49.2	26	17.8	14.2	11.5	9.9
dd-parallelBQA	98.4	48	23.8	16.2	12.7	10.3	8.6

Table 1: Running times of branch queries (in seconds $\times$ 10,000 queries).

5 Conclusion

Despite the vast amount of research done around wavelet trees, very little has been done to-date to optimize these data structures and their associated operations for current 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 $O(n)$-time construction and good use of memory resources. We also have shown a non-trivial parallelization of querying wavelet tree data.

In this paper we focused on the most general representation of a wavelet tree. However, some of our results may apply to other variants of wavelet trees. For example, it would be interesting to study how to extend our results to compressed wavelet trees (e.g., Huffman shaped *wtrees*) and to generalized wavelet trees (i.e., multiary wavelet trees where the fan out of each node is increased from 2 to $O(polylog(n))$). We shall explore also the extension of our results to the Wavelet Matrix [4] (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). We also plan to experiment with real data from other domains such as inverted indices [13], genome information and two-dimensional range searching (useful, for example, in position restricted substring searching) [15]. More future work also involves dynamization, whereby the *wtree* is being modified concurrently by many processes as it is queried.

After the last decades, it is evident that architecture has become relevant again. It is nowadays difficult to find single-core computers. It therefore 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, as we did here.[8]

[8] We would like to thank Diego Caro and Andrea Rodríguez for useful comments regarding previous versions of this paper.

References

1. Arroyuelo, D., Costa, V.G., González, S., Marín, M., Oyarzún, M.: Distributed search based on self-indexed compressed text. Inf. Process. Manage. 48(5), 819–827 (2012)
2. Blumofe, R.D., Leiserson, C.E.: Scheduling multithreaded computations by work stealing. J. ACM 46(5), 720–748 (1999)
3. Brisaboa, N., Luaces, M., Navarro, G., Seco, D.: Space-efficient representations of rectangle datasets supporting orthogonal range querying. Information Systems 35(5), 635–655 (2013)
4. Claude, F., Navarro, G.: The wavelet matrix. In: SPIRE. LNCS, vol. 7608, pp. 167–179 (2012)
5. Claude, F., Navarro, G.: Practical rank/select queries over arbitrary sequences. In: SPIRE. LNCS, vol. 5280, pp. 176–187 (2008)
6. Claude, F., Nicholson, P.K., Seco, D.: Space efficient wavelet tree construction. In: SPIRE. LNCS, vol. 7024, pp. 185–196 (2011)
7. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, chap. Multithreaded Algorithms, pp. 772–812. The MIT Press, third edn. (2009)
8. Faro, S., Külekci, M.O.: Fast multiple string matching using streaming SIMD extensions technology. In: SPIRE. LNCS, vol. 7608, pp. 217–228 (2012)
9. Ferragina, P., Manzini, G., Mäkinen, V., Navarro, G.: Compressed representations of sequences and full-text indexes. ACM Transactions on Algorithms 3(2) (2007)
10. Gagie, T., Navarro, G., Puglisi, S.J.: New algorithms on wavelet trees and applications to information retrieval. Theor. Comput. Sci. 426-427, 25–41 (2012)
11. González, R., Grabowski, S., Mäkinen, V., Navarro, G.: Practical implementation of rank and select queries. In: WEA (Poster). pp. 27–38. CTI Press (2005)
12. Grossi, R., Gupta, A., Vitter, J.: High-order entropy-compressed text indexes. In: SODA. pp. 841–850 (2003)
13. Konow, R., Navarro, G.: Dual-sorted inverted lists in practice. In: SPIRE. pp. 295–306. LNCS 7608 (2012)
14. Ladra, S., Pedreira, O., Duato, J., Brisaboa, N.R.: Exploiting SIMD instructions in current processors to improve classical string algorithms. In: ADBIS. pp. 254–267 (2012)
15. Mäkinen, V., Navarro, G.: Rank and select revisited and extended. Theor. Comput. Sci. 387(3), 332 – 347 (2007), the Burrows-Wheeler Transform
16. Makris, C.: Wavelet trees: A survey. Comput. Sci. Inf. Syst. 9(2), 585–625 (2012)
17. Matsumoto, M., Nishimura, T.: Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. ACM Trans. Model. Comput. Simul. 8(1), 3–30 (1998)
18. Navarro, G., Nekrich, Y., Russo, L.: Space-efficient data-analysis queries on grids. Theoretical Computer Science 482, 60–72 (2013)
19. Navarro, G.: Wavelet trees for all. In: CPM. LNCS, vol. 7354, pp. 2–26 (2012)
20. Otellini, P.: Keynote Speech at Intel Developer Forum. Internet (2003)
21. Raman, R., Raman, V., Satti, S.R.: Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. ACM Trans. Algorithms 3(4) (2007)
22. Sutter, H.: The free lunch is over: A fundamental turn toward concurrency in software. Dr. Dobb's Journal 30(3) (2005)
23. Tischler, G.: On wavelet tree construction. In: CPM. pp. 208–218 (2011)
24. Välimäki, N., Mäkinen, V.: Space-efficient algorithms for document retrieval. In: CPM. LNCS, vol. 4580, pp. 205–215 (2007)