Efficient Wavelet Tree Construction and Querying for Multicore Architectures

José Fuentes, Erick Elejalde, Leo Ferres,
Diego Seco, Andrea Rodríguez

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

Abstract. Wavelet trees have become very useful to handle large data 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 a practical multicore algorithm for wavelet tree construction that runs in $O(n)$ time using $\lg \sigma$ processors, and 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 [1, 2]: 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. For instance, the clock frequency of Intel's newest micro-architecture, Haswell, to be released in June 2013, ranges from 2GHz to 3.9GHz. 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 (both asymptotically and practically) 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* [3], which can be seen as a way to represent a sequence, a reordering of elements, or a grid of points. Wavelet trees have been useful in a wide range of problems, from representing and querying sequences and documents in information retrieval [4, 5], determining permutations for generic numerical sequences [6], and solving geometric problems such as visibility of points in grids [7].

Any application for a wavelet tree is an application for a parallel wavelet tree. However, one can think of several practical applications especially suited for parallel wavelet trees. In particular, those that handle so-called big or massive data. An example of such type of applications is searching gene encoding [8], where not only the data to search in (e.g., a complete genome sequence), but also the

data to search for (e.g., the recognized genes) is large. There are 16 million genes that are already recognized and stored in a database such as genBank[1]. Another example is positional inverted indexing for document retrieval, an application of wavelet trees to represent sequences which continuously demands indexing larger text collections [9]. This is to name but a few applications. For a comprehensive and up-to-date overview, see [3].

Our contributions in this paper are as follows: we first propose a linear $O(n)$ time parallel algorithm for wavelet tree construction using $\lg \sigma$ processors (Section 3.1)[2] and report experiments showing the algorithm to be practical for large datasets and alphabets, achieving close to perfectly linear speedup (Section 4.1). In order to exploit multicore architectures, we also investigated techniques to speed up range queries and propose BQA, a hybrid domain-decomposition/parallel batch processing technique (Section 3.2) 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 (Section 4.2). To the best of our knowledge, this is the first proposal of a parallel wavelet tree (in the dynamic multithreading model, see Section 2.2).

2 Preliminaries

2.1 Wavelet trees

Since Grossi et al. introduced the wavelet tree in 2003 [10], this compact data structure has spurred much research (see [3, 11] for comprehensive surveys). Although it has been original devised as a data structure for encoding a reordering of the elements of a sequence, it has been successfully used in many applications. For example, they have been used to index sequences [10, 12, 13], documents [14], grids [15] and even sets of rectangles [16], to name just a few applications.

The wavelet tree presents some nice theoretical properties. For example, it can be stored in space bounded by different measures of the entropy of the underlying data, thus enabling compression. But, in addition, they are also efficiently implementable [17] and perform well in practice. For the purpose of this paper, let us describe the wavelet tree in a simple and general way as 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 appear 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.

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 $R_l \subset R$ and the right child a subset $R_r = R \backslash R_l$. Every node virtually represents a subsequence S' of S composed of symbols

[1] `http://www.ncbi.nlm.nih.gov/genbank`

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

whose value lies in R. This subsequence is stored as a bitmap of length n, and, for each position i in the bitmap, 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 (see for example [18]). Its construction takes $O(n \lg \sigma)$ time (we do not consider space-efficient construction algorithms [19, 20]). As we 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 [21], which is important for large alphabets. In favor of clarity, we focus on the simple form, though our results can be extended to other encodings and tree shapes.

Besides from the basic primitives described above, the wavelet tree supports richer queries than initially imagined. For example, Mäkinen and Navarro [21] 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 [22], the authors represent posting lists in a *wtree* and solve ranked AND queries by synchronized solving of several range queries. 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 $s_x \in [j, j']$[3]. The counting version of the problem can be defined analogously.

Some work has been done in parallelism. In [9], authors explore the use of wavelet trees in distributed web search engines. They assume a distributed memory model and propose two partition techniques (document- and term-based) to balance the workload of the 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 [23], authors explore the use of SIMD instructions to improve the performance of wavelet trees (and other string algorithms, see, for example, [24]). 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.

2.2 Dynamic multithreading

Dynamic multithreading (henceforth, DYM) [25] is a model of parallel computation faithful to several industry standards such as Intel's CilkPlus[4] and Threading Building Blocks[5], OpenMP Tasks[6], and Microsoft's Task Parallel Library[7].

[3] Notice that a grid is a particular case where $\sigma = n$.
[4] `http://cilkplus.org/`
[5] `http://threadingbuildingblocks.org/`
[6] `http://openmp.org/wp/`
[7] `http://msdn.microsoft.com/en-us/library/dd460717.aspx`

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.[8] 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 [26].

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)[9], 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 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

3.1 Parallel construction

In favor of clarity, we focus on binary wavelet trees where the symbols of Σ are contiguous in $[1, \sigma]$. If they are not contiguous, we can use a bitmap to remap the sequence to a contiguous alphabet [17]. 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[10], can build a *wtree* in $T_1 = O(n \lg \sigma)$ time by a linear processing of the symbols at each node. The recursive `rwt` algorithm works by halving Σ recursively into binary sub-trees whose left-child are all 0s and the right all 1s, until 1s and 0s mean only one symbol in Σ. We parallelized `rwt` by the

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

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

[10] `http://libcds.recoded.cl`

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

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
```

usual technique of **spawn**ing one task for each recursive call except the last, while doing the latter on the calling thread [27]. In our case, we **spawn** the left sub-tree to continue working on the right sub-tree. As we show in Section 4, this naive parallel version of the recursive algorithm does not behave well in practice. Indeed, in what follows we analytically show that it does not get any parallelism in the worst case.

The DAG of the `rwt` is weighted. Not all strands in this DAG are the same weight: the frequency of symbols is not the same. All paths are the same length; that is, $O(\lg \sigma)$, the critical path will be given by the weight of the heaviest path in the DAG. In the worst case, where one branch always contains most of S, $T_\infty = O(n \lg \sigma)$. This is the case, for example, when Σ is ordered by frequency. In the best case, when all symbols in Σ have exactly the same frequency, then $T_\infty = O(n)$. Finally, the parallelism for the worst case of `prwt` is $T_1/T_\infty = O(1)$, which is no parallelism at all. In turn, in the best case, we have that the parallelism is $O(\lg \sigma)$, which means that the algorithm scales on 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. Therefore, 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 for each symbol of the original sequence its node at level i. The following proposition shows how it can be computed.

Proposition 1. *Given a symbol $s \in S$ and a level i, $0 \leq i < l$, of a wtree with $l = \lceil \lg \sigma \rceil$ levels, we can compute the node at which s is represented at level i 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 Section 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 the number of levels; i.e., $\lceil \lg \sigma \rceil$ (line 1). 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 the entire level 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 the first half symbols represented at the node (and stores a 0 bit), or to the second half (and stores a 1 bit). 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 the level. Finally, the bitmaps in this structure need to support rank/select operations, thus the construction algorithm must create the 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 becomes exponentially larger when more levels are created, until we reach σ 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 which the work-stealing scheduler will balance (see Section 2.2).

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 bibliography 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, which improves on the worst case of the recursive algorithm. 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 the optimal speedup.

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

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  |  if batch_q.x^s > batch_q.x^e ∨ (isLeaf(WT_i^j) ∩ batch_q.y_range) = ∅ then
8  |  |  local_states_q ← 1
9  |  |  continue
10 |  if isLeaf(WT_i^j) then
11 |  |  results_q^j ← batch_q.x^e − batch_q.x^s + 1      /* j is the label */
12 |  |  continue
13 |  lbatch_q.x^s ← rank_0(WT_i^j.bitmap, batch_q.x^s − 1) + 1
14 |  lbatch_q.x^e ← rank_0(WT_i^j.bitmap, batch_q.x^e)
15 |  rbatch_q.x^s ← rank_1(WT_i^j.bitmap, batch_q.x^s − 1) + 1
16 |  rbatch_q.x^e ← rank_1(WT_i^j.bitmap, batch_q.x^e − 1)
17 |  lbatch_q.y_range ← batch_q.y_range
18 if ∀_q, local_states_q = 1 then return
19 spawn parallelBQA(WT_{i+1}^{2j}, lbatch, num_queries, local_states, results)
20 parallelBQA(WT_{i+1}^{2j+1}, rbatch, num_queries, local_states, results)
21 return                                  /* implicit sync */
```

3.2 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* [28].

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[11].

We implemented two branch-query-answering techniques: *individual*-query-answering (IQA) and *batch*-query-answering (BQA). The IQA technique is the

[11] For lack of space, we do not report this simple experiment, but it follows exactly the same methodology reported in Section 4.2.

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)[11].

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** were to be 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 with no compiler optimizations. 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>`[12]

4.1 Construction experiments

We compared the implementation of our parallel wavelet tree construction algorithm (henceforth *pwt*) against a naively parallelized version *prwt* of the traditional $O(n \lg \sigma)$ recursive algorithm *rwt*, the latter sequential algorithm being our baseline for comparison. 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[13] 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

[12] This is a reproducible-research-friendly paper, everything needed to replicate these results is available at REDACTED FOR BLIND REVIEW

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

was ordered by their frequency of occurrence in the original `english` text. For experimentation, we generated two kinds of alphabets Σ' of size 2^k: the first was constructed by taking the top 2^k *most frequent* words in the original Σ, while the second did away with the frequency information, and assigned each symbol a random index using a Marsenne Twister [29] on the frequency alphabet, with $k \in \{4, 6, 8, 10, 12, 14\}$ in both cases. 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' ($\sim$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)[14].

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 network tasks. We also report the speedup as defined above in Section 2.2, sequential time divided over parallel time on P processors.

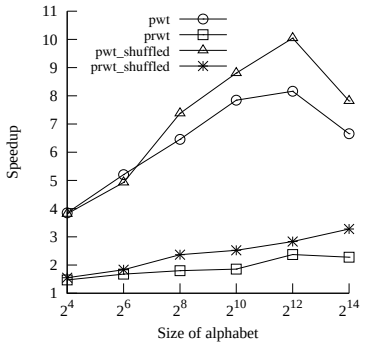

Fig. 1: Speedup over σ with a 2GB text.

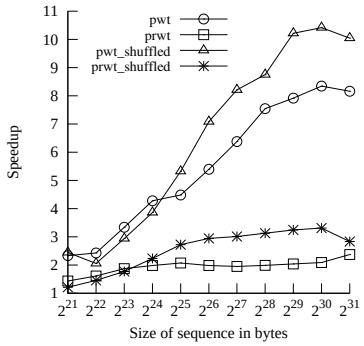

Fig. 2: Speedup over n, with $\sigma = 2^{14}$.

Figure 1 shows this speedup ($P = 12$) for both implemented parallel algorithms: the `prwt` (based on the traditional recursive algorithm), and `pwt`, our iterative parallel implementation. We also show the behavior of these algorithms when the alphabet is shuffled (random) versus ordered by frequency of occurrence in the original text. The iterative implementation of `pwt` shows good parallel behavior, scaling linearly over $\lg \sigma$ processors. The `pwt` algorithm has $O(\lg \sigma)$ parallelism; thus, speedup does not improve beyond $P > \lg \sigma$, but neither does it get worse. Thus, although we report speedup for $P = 12$, the same speedup for each σ is reached at $P = \lg \sigma$. Readers may notice a drop in speedup at

[14] 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.

$\sigma = 2^{14}$, but this is because the hardware had 12 processors. Upon reaching this point, the machine cannot create any more *wtree* levels in parallel and resorts to scheduling the excess tasks on the available processors, slowing down the general construction process.

A bit more surprising is that `pwt` also seems to scale over n (see Figure 2). This is a nice side effect of our parallel algorithm that does not happen with `prwt`. This speedup is due to the fact that we expect larger values of n and σ to offset the time taken to create and schedule worker threads and the tasks mapped onto those threads. In the case of `prwt`, the generation of a task per spawned *wtree* node makes scheduling and administration costlier, affecting scalability.

4.2 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 Figure 3). We repeated each experiment three times. We worked with the maximum throughput taken by all three repetitions, similar to Section 4.1.

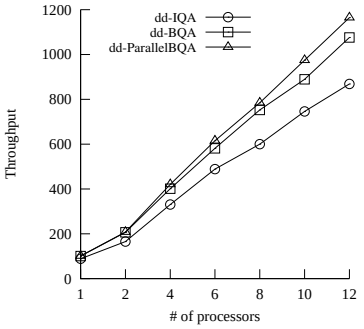

Fig. 3: Throughput over P for 100 batches of 100 queries (10,000 queries).

	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).

As discussed in Section 3.2, 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. To see the orders of magnitude of the implementations' running times, see Table 1.

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*, and 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. In the full version of this article we shall present how to extend our results to compressed wavelet trees (for example, 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 [30] (a different levelwise 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 [22], genome information and grids. More future work also involves dynamization, whereby the *wtree* is being modified concurrently by many processes as it is queried.

After two 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.

References

1. Otellini, P.: Keynote Speech at Intel Developer Forum. Internet (September 2003)
2. Sutter, H.: The free lunch is over: A fundamental turn toward concurrency in software. Dr. Dobb's Journal **30**(3) (March 2005)
3. Navarro, G.: Wavelet trees for all. In: CPM. Volume 7354 of LNCS. (2012) 2–26
4. Barbay, J., Gagie, T., Navarro, G., Nekrich, Y.: Alphabet partitioning for compressed rank/select and applications. In: ISAAC. Volume 6507 of LNCS. (2010) 315–326
5. Belazzougui, D., Navarro, G.: Alphabet-independent compressed text indexing. In: ESA. Volume 6942 of LNCS. (2011) 748–759
6. Barbay, J., Navarro, G.: Compressed representations of permutations, and applications. In: STACS. Volume 3 of LIPIcs. (2009) 111–122

7. Navarro, G., Russo, L.M.S.: Space-efficient data-analysis queries on grids. In: ISAAC. Volume 7074 of LNCS. (2011) 323–332
8. Schnattinger, T., Ohlebusch, E., Gog, S.: Bidirectional search in a string with wavelet trees and bidirectional matching statistics. Inf. Comput. **213** (2012) 13–22
9. Arroyuelo, D., Costa, V.G., González, S., Marín, M., Oyarzún, M.: Distributed search based on self-indexed compressed text. Inf. Process. Manage. **48**(5) (2012) 819–827
10. Grossi, R., Gupta, A., Vitter, J.: High-order entropy-compressed text indexes. In: SODA. (2003) 841–850
11. Makris, C.: Wavelet trees: A survey. Comput. Sci. Inf. Syst. **9**(2) (2012) 585–625
12. Ferragina, P., Manzini, G., Mäkinen, V., Navarro, G.: Compressed representations of sequences and full-text indexes. ACM Transactions on Algorithms **3**(2) (2007) article 20
13. Chien, Y.F., Hon, W.K., Shah, R., Vitter, J.S.: Geometric Burrows-Wheeler transform: Linking range searching and text indexing. In: DCC. (2008) 252–261
14. Välimäki, N., Mäkinen, V.: Space-efficient algorithms for document retrieval. In: CPM. Volume 4580 of LNCS. (2007) 205–215
15. Navarro, G., Nekrich, Y., Russo, L.: Space-efficient data-analysis queries on grids. Theoretical Computer Science **482** (2013) 60–72
16. Brisaboa, N., Luaces, M., Navarro, G., Seco, D.: Space-efficient representations of rectangle datasets supporting orthogonal range querying. Information Systems **35**(5) (2013) 635–655
17. Claude, F., Navarro, G.: Practical Rank/Select Queries over Arbitrary Sequences. In: SPIRE. (2008) 176–187
18. Raman, R., Raman, V., Rao, S.S.: Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In: SODA. (2002) 233–242
19. Claude, F., Nicholson, P.K., Seco, D.: Space efficient wavelet tree construction. In: SPIRE. (2011) 185–196
20. Tischler, G.: On wavelet tree construction. In: CPM. (2011) 208–218
21. Mäkinen, V., Navarro, G.: Rank and select revisited and extended. Theoretical Computer Science **387**(3) (2007) 332–347
22. Konow, R., Navarro, G.: Dual-sorted inverted lists in practice. In: SPIRE. LNCS 7608 (2012) 295–306
23. Ladra, S., Pedreira, O., Duato, J., Brisaboa, N.R.: Exploiting SIMD instructions in current processors to improve classical string algorithms. In: ADBIS. (2012) 254–267
24. Faro, S., Külekci, M.O.: Fast multiple string matching using streaming simd extensions technology. In: SPIRE. Volume 7608 of LNCS. (2012) 217–228
25. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Multithreaded Algorithms. In: Introduction to Algorithms. Third edn. The MIT Press (2009) 772–812
26. Blumofe, R.D., Leiserson, C.E.: Scheduling multithreaded computations by work stealing. J. ACM **46**(5) (September 1999) 720–748
27. Leiserson, C.E.: The cilk++ concurrency platform. In: DAC, ACM (2009) 522–527
28. Gagie, T., Navarro, G., Puglisi, S.J.: New algorithms on wavelet trees and applications to information retrieval. Theor. Comput. Sci. **426-427** (2012) 25–41
29. Matsumoto, M., Nishimura, T.: Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. ACM Trans. Model. Comput. Simul. **8**(1) (1998) 3–30
30. Claude, F., Navarro, G.: The wavelet matrix. In: SPIRE. Volume 7608 of LNCS. (2012) 167–179