Bit-Parallel (Compressed) Wavelet Tree Construction

Video Script

Introduction

Hello, I am Patrick Dinklage from the technical university of Dortmund, Germany. I am going to talk about the bit-parallel construction of wavelet trees and compressed wavelet trees. This is joint work with Johannes Fischer, also from Dortmund, Florian Kurpicz from the Karlsruhe institute of technology, and – most importantly – Jan-Philipp Tarnowski. He now works at crownpeak, but he wrote his master's thesis with us and essentially, this work is a round-up of his practical results.

Wavelet Trees (Introduction)

Let us begin by defining wavelet trees. We are given an input text, `wavelet_tree` as an example, over an alphabet shown on the right hand side. To each character, we assign a binary code. In this case, the codes are simply increasing binary codes of the characters in lexicographic order. Because the alphabet has size 8, we use 3 bits to encode each character.

We now build a bit vector that is as long as the text, and assign to every position the most significant bit of the character's code. This labels the root node of the wavelet tree. We now split up the text into two child nodes: the left node represents characters whose most significant bit is 0, and the right child represents the remaining characters whose most significant bit is 1.

We then proceed recursively, using the second most significant bit on the second level of the tree and so forth, until we have computed all $\lceil \lg \sigma \rceil$ levels, where σ is the size of the alphabet. On every level, we end up with exactly n bits.

Wavelet Trees (Bits Only)

We only keep these bits and discard the texts. Because we only rearranged the bits of the characters in the order in which they occur in the text, the text can still be decoded.

Wavelet Trees (Levelwise)

To save space, we also remove the pointers between nodes and their children, and concatenate the bit vectors on each level. The tree structure is still implicitly there. For example, the number of 0-bits in the root node gives us the number of bits in the left child. By keeping track of node boundaries, and using constant-time rank queries on the bit vectors, we can still navigate the implicit tree without any pointers.

The wavelet tree thus fits into $\lceil \lg \sigma \rceil (n + o(n))$ bits and is an important ingredient for compressed text indexing, a popular example being the FM index. Its main application is arguably to answer integer rank and select queries, but it is also commonly used in geometric data structures.

Wavelet Tree Construction

For this reason, there has been quite some research on constructing the wavelet tree for a given text efficiently. Straightforward text book algorithms already achieve a construction time $\mathcal{O}(n \lg \sigma)$, matching the output size. The *prefix counting* algorithm works very well in practice. Not only can it be applied in all kinds of computational models – which we do in the referred 2022 journal article – but it is also the fastest sequential algorithm in practice known thus far.

Independently from one another, in 2014, Babenko et al. and Munro et al. squeezed out a $\sqrt{\lg n}$ factor of the running time using word packing techniques. This was implemented by Kaneta in 2018 using vector CPU instructions, with promising results. However, they only did this for commonplace 64-bit architectures.

For this work, we posed the following two questions:

First, does Kaneta'a algorithm scale with the register size? We investigated this for registers up to 512 bits, powered by the AVX-512 instruction sets.

Second, can Kaneta's algorithm be adapted to construct also the Huffman-shaped, or compressed, wavelet tree that we will have a closer look at later in this video?

Fast Levelwise Construction (Introduction)

To approach these questions, let us have a closer look at the fast construction algorithm. For a meaningful visualization, we consider a larger alphabet of size 256, meaning every character in the text is encoded using 8 bits. Let us look at a sequence in the text consisting of the letters *B*, *C*, *D* and *A* that have the given codes.

Our first step is to compute the generalized wavelet tree of arity 2^τ, where τ is a parameter that we set to 4 in this example. In the generalized tree, instead of assigning bits to each text position, we assign integers of τ bits. Apart from this, the structure remains the same. Consequently, the multiary wavelet tree consists of only $\lceil \lg \sigma \rceil / \tau$ levels.

Fast Levelwise Construction (Clusters)

Observe how each level corresponds to τ levels in the binary wavelet tree. We call these *clusters* in the following.

Our grand strategy is as follows. We expand each cluster to the corresponding τ levels of the binary wavelet tree in linear time. If we can do that, then doing it for all $\lceil \lg \sigma \rceil / \tau$ clusters takes time $\mathcal{O}(n \lg \sigma / \tau)$. By setting τ to $\sqrt{\lg n}$, which we will also need later, we match the desired total running time.

Fast Levelwise Construction (Steps)

So we are allowed to spend at most $\mathcal{O}(n)$ time for each cluster. Let us look at the steps that we undertake, which we will have a closer look at in the following.

The first step called *Cluster Extraction* consists of a preliminary scan over the text and takes linear time. The magic happens in step 2: we do τ passes over the cluster, and each pass consists of the substeps *Bit Extraction* and *List Splitting*, taking time $\mathcal{O}(n\tau / \lg n)$ each. Here, the choice of $\tau = \sqrt{\lg n}$ comes in: over all τ passes, the quotient $\tau^2 / \lg n$ cancels out and we get linear time for the τ passes. The last step, *Text Reshuffling*, again consists of a linear-time scan preparing the next cluster.

Cluster Extraction

Let us begin with the first step, *Cluster Extraction*. In this step, we extract from each character in the text the block of τ bits relevant for the current cluster. In this example, we are in the first cluster, hence we extract the τ most significant bits from each character, which are underlined. We store these in a *packed list* that we call S. S is an array of computer words of w bits – 16 bits in this example – each containing τ-bit blocks of multiple characters.

Using standard word operations – namely ANDs, ORs and shifts – we can easily do all this in linear time, assuming the word RAM model.

Steps (after Cluster Extraction)

To this end, we check off the *Cluster Extraction* step.

We now proceed to expanding the cluster to the τ bit vectors in the binary wavelet tree, and we do this in τ passes.

Word Packing

Let us first look at the key ingredient for this, which is word packing. In the packed list S, we pack w/τ blocks into each word, where w is the word size, 16 bits in our case. Therefore, S consists of $n\tau/w$ words.

If we take constant time to process one word, then processing all of S takes time $\mathcal{O}(n\tau/w)$. With w being $\Omega(\lg n)$, we get exactly what we want. This is the key idea to the accelerated algorithm where the speedup comes from processing a word-packed list.

So, in each pass, we need to take care to only do constant work for every word in S.

Bit Extraction

Let us first look at Bit Extraction. From the τ-bit blocks of each character that we stored in the S, we extract those bits that are relevant for the current level in the binary wavelet tree. Thanks to word packing, we can extract w/τ bits at a time as follows.

We build a bitmask, let's call it M, that we store in a computer word. In M, we mark every bit that we want to extract from the word in S with a 1, and leave the remaining bits at zero. Let's say we are in pass number t, then in M, we set the t-th bit for every block of τ bits. The marked bits are exactly those that we need to construct level $\alpha\tau + t$ of the binary wavelet tree, where α denotes the number of the cluster that we are in.

Using a specialized instruction, we can extract the bits marked by M and align them in the most significant bits of the output word in constant time. In theory, this can be done via lookup tables. In practice, we have the *parallel bits extract* instruction, short `pext`, or variants thereof depending on the register size.

Steps (after Bit Extraction)

This was the most important part regarding the computation of the wavelet tree. Next, the *List Splitting* step prepares the next pass.

List Splitting (Preface)

First, why is this step needed? Recall how we split the text at a wavelet tree node: characters with a 0-bit are moved to the left child, and those with a 1-bit are moved to the right child. This means that the order of the characters' occurrences in the text changes, and this must be reflected in our packed list S after every pass.

List Splitting (Borders)

For this, observe how we can compute a node boundary, depicted here as the thick red line, simply by counting the number of 0-bits in the parent. By counting 0-bits during the Bit Extraction that we just looked at earlier, we can compute all node borders for the wavelet tree level on the fly.

We must take care here and make use of operations that can count the number of 0-bits in a word in constant time, otherwise we break our guarantees. This can be done using the commonplace `popcount` instruction.

List Splitting (Right Child)

Consider the bits that we just extracted from the example part of S during the previous *Bit Extraction* step. Let us expand each bit to τ copies of that bit. With lookup tables, we can do this in constant time. The result is a mask M that marks precisely those blocks that we want to move to the right child, and we can again use `pext` to extract them.

List Splitting (Left Child)

To do the same for the left child, we simply flip the bits.

Doing `pext` twice still only takes constant time, and so we are good regarding our guarantees.

Steps (after List Splitting)

After doing *Bit Extraction* and *List Splitting* for all of the τ passes, all of the work is done for the cluster.

However, we are left to do some preparation before proceeding to the next cluster.

Text Reshuffling

The reason is similar to the reason for which we needed the List Splitting. Recall how the text is split, this time at a node in the generalized wavelet tree. Each character is assigned a block of τ bits. Conveniently, these are the binary representation of the number of the child node that the character is moved to. Before going to the next cluster, we need to rearrange the text accordingly.

This is very simple, however. In essence, we are doing a stable counting sort, and the node borders are already known, because we kept track of them during the *Bit Extraction* passes. Clearly, the rearrangement can be done in a left-to-right scan over the text.

Steps (after Text Reshuffling)

This concludes the improved algorithm. We showed that we need linear time to process a cluster, and since there are $\lceil \lg \sigma \rceil / \tau$ clusters, with τ set to $\sqrt{\lg n}$, the total construction time becomes $\mathcal{O}(n \lg \sigma / \sqrt{\lg n})$.

Now, let us tackle one of our initial questions: can this be done for Huffman-shaped Wavelet Trees?

Huffman-Shaped Wavelet Trees (Huffman Tree)

As the name suggests, we look at the Huffman tree for our text, which looks like this. We now use the Huffman codes to construct the wavelet tree.

Huffman-Shaped Wavelet Trees

When we do that, the wavelet tree consists of the inner nodes of the Huffman tree, hence the name *Huffman-shaped* wavelet tree. In this regard, this is an entropy-compressed version of the wavelet tree.

However, what happens if we try to put it in the levelwise representation?

Huffman-Shaped Wavelet Trees (Gaps)

We see that there are gaps caused by missing nodes. These come from Huffman codes of more frequent characters being shorter than others. These are problematic, because the bit vectors are no longer consecutive, and that breaks our argument that the number of 0-bits in a node equals the number of bits in the left child. As a consequence, the navigation in the implicit tree using constant-time rank queries no longer works.

Luckily, there is a solution. From the Huffman table, we can compute so-called canonical Huffman codes. These are still optimal, but reshape the Huffman tree so that all gaps are moved to the left.

Huffman-Shaped Wavelet Trees (Inverted Canonical)

By inverting the canonical Huffman codes, we move all the gaps to the right. That way, our requirements for navigation are met again.

In our construction algorithm, we still somehow need to account for codes that end in some pass processing a cluster.

Fast Huffman-Shaped Construction (Steps)

To do this, we add two new steps.

First, when extracting the relevant τ-bit blocks for the cluster, we also compute code lengths, for which we spend linear time.

Second, before the *List Splitting* step, we insert a new step called *List Filtering* to each of the τ passes, allowing only constant time to be spent for each word in the packed list.

Code Length Computation

When preparing the cluster, additionally to the packed list S, we also compute a packed list L. For each character, it contains the remaining code length within the cluster. In our example here, let's say that the Huffman codes of *B* and *C* have 6 bits; and those of `D` and `A` only have 5 bits. Say we are in the second cluster of τ bits, then we store 2, 2, 1, 1 in L, because that's how many bits are left for the cluster.

We artificially limit the lengths to τ, simply so that each length is guaranteed to fit into τ bits. With this, the entries of L are aligned with those in S containing the τ-bit blocks.

If we have constant-time access to the Huffman code table, we can clearly compute L in time $\mathcal{O}(n)$.

List Filtering

Now we are in pass number t processing the cluster. Some codes may end at the current level of the binary wavelet tree, $\alpha\tau + t$, and we want to filter these out before going for the next pass.

We can find ending codes by doing a parallel greater-than comparison of the τ entries in L against τ copies of t. These fit into a computer word, because t is at most τ. For the comparison, there are instructions that, in constant time, give us a mask M as a result, marking all entries where indeed, the value in L is greater than t. As you may guess, we use M as an input for the `pext` instruction in order to filter out the characters whose codes end at the current level, and we can do so in constant time.

If we find that any code ends in the current word, then we can conclude that all remaining codes also end for the remainder of the text. This is because the inverted canonical Huffman codes moved all gaps to the right.

It should be noted that L needs to be filtered and split just like S, but this does not change the overall constant work for each word.

Steps (after Huffman)

With the new steps, the algorithm is fit to construct the Huffman-shaped wavelet tree with no asymptotic penalties. This answers one of our main questions.

Useful CPU Instructions

We implemented different versions of the construction algorithm. For this, we made use of a variety of useful CPU instructions based on the register size and what we found works best for a given task. This table gives only a brief overview; it would take a while to explain these in detail, but we already mentioned some of these earlier in the video. For a documentation, we refer to the Intel Intrinsics Guide. In our paper, we explain where and how we use these instructions.

Experimental Results

Let us look at our experimental results.

We plot here the throughputs of different versions of our algorithm against `pc`, which stands for *prefix counting*, the fastest known sequential algorithm thus far. The different versions of our algorithm use different register sizes, as stated in their names. The algorithm `ext64` uses the `pext` instructions for basically everything as was shown in the video. The `shuf` algorithms use

different instructions depending on the register size, but they compute the same intermediate results.

We let the algorithms compute the Huffman-shaped wavelet trees for different files from the Pizza & Chili corpus, as well as three additional, much larger input files. These are a web crawl, a long DNA sequence and a Wikipedia XML dump, respectively. Our hardware has an Intel Rocket Lake CPU that covers most of the different AVX-512 instruction sets, featuring register sizes up to 512 bits.

The results tell us two things.

First, indeed, the algorithm scales very well as the register size increases. By using 512-bit registers, we always achieve the fastest wavelet tree construction.

Second, we outperform *prefix counting*, the fastest sequential algorithm known thus far, for every input. This holds even for `ext64`, which uses commonplace 64-bit registers and can be used on almost any current hardware.

Our source code is written in C++ and is published on Github at the given URL under an open source license.

Thank you very much for your attention.