Parallel and Vectorized Wavelet Tree Construction

Master's Thesis of

Marcel Hollerbach

at the Department of Informatics
Theoretical Informatics
Karlsruhe Institute of Technology

Reviewer: Prof. Dr. Peter Sanders

Advisor: Dr. Florian Kurpicz

01. September 2021 – 01. March 2022

Karlsruher Institut für Technologie
Fakultät für Informatik
Postfach 6980
76128 Karlsruhe

I declare that I have developed and written the enclosed thesis completely by myself, and have not used sources or means without declaration in the text.

PLACE, DATE

. . . .

(Marcel Hollerbach)

Abstract

The wavelet tree [13] is a fast look up structure for strings over an alphabet $[0, \sigma)$ and the length n. It can be used to answer *rank*, *select*, and *access* queries in $O(\log \sigma)$ time. In this work, the algorithms of Kaneta [9] are implemented. This implementation is based on the work of Babenko et al. [10] and Munro et al. [4], they showed that wavelet trees could be build in $O\left(n \log(\sigma)/\sqrt{\log(n)}\right)$ by utilizing table look ups to process entire words. Kaneta implemented this approach using special instruction like *pext*, and *pshufl*, which replaced the need of creating tables. Additionally, parallel versions are implemented by using domain decomposition. As an additional parallel construction, the approach of [8] is implemented. As before, the table look ups are replaced with *pext* instructions.
All the construction algorithms implemented are then compared to PWM from Dinklage et al. [2], which require $O(n \log(\sigma))$ time for construction. In the comparison of PWM with the sequential algorithms, the *pext* tree creation were able to outperform the PWM algorithms for $\log \sigma > 3$. For matrix creation, they only performed better for $\log \sigma > 5$. Comparing the parallel approaches of domain decomposition and the approach of Shun showed, that domain decomposition performs a lot better compare to Shun. Comparing the domain decomposition of PWM with the one of the *pext* algorithms showed that the *pext* versions are outperforming the PWM versions. This time only for $\log \sigma = 8$ for the biggest payloads. And $\log \sigma > 6$ for medium sized payloads. Additionally to that, the relation between τ and $\log \sigma$ is getting explained. Finally, a new alternative approach to the algorithm from Babenko et al. and Munro et al. is described.

Zusammenfassung

Wavelet trees [13] werden als Index für Zeichenketten über ein $[0, \sigma)$ und einer Länge n genutzt. Der Index wird genutzt um *rank*, *select* und *access* Anfragen zu beantworten. Diese Anfragen beantworten wie oft ein Buchstabe vorkommt, wo der n-te Buchstabe in der Zeichenkette vorkommt, oder welcher Buchstabe an einer bestimmten Position steht. All diese Anfragen können in $O(\log \sigma)$ Zeit beantwortet werden. In den Arbeiten von Babenko et al. [10] und Munro et al. [4] wurde erstmals ein ansatz gezeigt, indem man Wavelet Strukturen in $O\left(n\log(\sigma)/\sqrt{\log(n)}\right)$ Zeit Konstruieren kann. Dieser Ansatz nutzt dabei Tabellen, um mehrere Bits auf einmal zu verarbeiten. In der Arbeit von Kaneta [9] werden diese Tabellen dann durch Instruktionen wie *pext* und *pshufl* ersetzt. Das resultiert darin, dass die theoretisch benötigte Zeit von $O(n\log(\sigma))$ auf $O\left(n\log(\sigma)/\sqrt{\log(n)}\right)$ verringert wird. In dieser Arbeit werden diese Algorithmen mit *pext* zunächst erneut implementiert. Dann werden diese durch Domain Decomposition parallelisiert. Zusätzlich wird ein weiterer Ansatz von Shun [8] implementiert. Dieser bildet eine Alternative zur Domain Decomposition. Im zweiten Teil der Arbeit werden dann die sequentiellen Algorithmen von Dinklage et al. [2] mit den *pext* Versionen verglichen. Dabei zeigt sich, das die *pext* Algorithmen für $\log \sigma > 3$ schneller als diese in PWM sind. Die Matrix zu erstellen ist ebenfalls schneller für $\log \sigma > 5$. Der Vergleich der beiden parallelen Versionen von Domain Decomposition und dem Ansatz von Shun zeigt, dass die Domain Decomposition wesentlich schneller ist, als die von Shun. Die Domain Decomposition wird dann mit den parallelen Algorithmen von PWM verglichen. Dabei zeigt sich, dass hier die *pext* Versionen bei großen Eingaben für $\log \sigma > 8$ schneller als die PWM Algorithmen sind. Bei mittleren Eingabegrößen sogar für $\log \sigma > 6$. Zusätzlich wird in der Arbeit der Zusammenhang zwischen τ und $\log \sigma$ erklärt. Letztlich wird noch ein neuer Ansatz vorgestellt, welcher die Idee von Babenko et al. und Munro et al. aufgreift, und diese leicht verändert.

Contents

Abstract i

Zusammenfassung iii

1. Introduction 1
- 1.1. Related Work 1
- 1.2. Wavelet Tree and Wavelet Matrix 2
  - 1.2.1. Rank / Select Queries on Wavelet Trees and Matrices 4
- 1.3. Additional Instructions 5
- 1.4. Algorithem Parallelization 5
- 1.5. Contribution 7

2. Sequential Algorithms 9
- 2.1. Bignode Tree Building 9
- 2.2. Creating Wavelet Tree and Wavelet Matrix out of Bignode Tree 11
  - 2.2.1. Wavelet Tree Construction 14
  - 2.2.2. Wavelet Matrix Construction 16
  - 2.2.3. Influence of τ 17
  - 2.2.4. Runtime 17
- 2.3. Dynamic τ Adjustment 18

3. Parallel Algorithms 21
- 3.1. Domain Decomposition 21
- 3.2. Shun Parallelization 23

4. Evaluation 29
- 4.1. Sequential Runtime of Basic Settings 30
- 4.2. Comparing Domain Decomposition Mergers 32
- 4.3. Comparing Shun and Domain Decomposition 32
- 4.4. Comparing to PWM 35
  - 4.4.1. Comparing using Real World Data 36
  - 4.4.2. Comparing using Generated Data 40
- 4.5. Memory Consumption 42
- 4.6. Intrinsic Functions 44
- 4.7. Code-Size 46
- 4.8. Runtime prediction for dynamic τ setting 46

5. Conclusion 49
- 5.1. Conclusion over *pext* algorithms 49

5.2. Further Improvements 50

Bibliography **51**

A. Appendix **53**

A.1. Parallel wavelet construction using generated data 54

A.2. Histograms of example texts 57

List of Figures

1.1. Visualization of a wavelet tree for $\log \sigma = 4$. Each cell has always 2 children, and characters initially in the cell, must end up in one of its children. Additionally, the length of all cells of a level is always n. 3
1.2. Visualization of a wavelet matrix for $\sigma = 4$. Each cell has always 2 children, and characters initially in the cell, must end up in one of its children. The big difference to the tree is here, that the order of children cells is different. 3
1.3. The wavelet tree of the word *hello world*. The vertical lines are showing where each cell starts or ends. Not each line fills all available cells, hence the number of visible cells is not always the maximum that is possible. . 4
1.4. The wavelet matrix of the word *hello world*. The vertical lines are separating the 0s cells from the 1s cells. If one cell is empty, no vertical line is added. 4
1.5. Explanation of the *pext* semantics. A/B/C/D/E/F/G/H are placeholders for bits. 5

2.1. Overview showing the relation between input, first phase, second phase, bignode tree, and resulting wavelet tree. The arrows of the first phase display that these values are represented in the associated line. The arrows of the second phase show in which lines the *pack* operation results are stored. 9
2.2. Example bignode tree for in the input sequence *hello world* and $\tau = 2$. . . 11
2.3. Constant values ℓ and h for $\tau = 4$, which are used for *split&sort* and *pack* operations. 12
2.4. Subtraction result of a block with a MSB not being set 13
2.5. Subtraction result of a block with a MSB being set 13
2.6. Results of the *splitsort(0, input)* call. L_0 contains all blocks that have a 0 at block position 0. L_1 contains all blocks that have a 1 at block position 1. Each result has two blocks of result. 14
2.7. Results of a *pack(1, input)* call. 14
2.8. An example histogram line, and bignode line at $i = 2$. The bignode tree line is stored using $\beta = 8$. There are 4 cells in total, which are stored in 2 8-bit long words. 15
2.9. Wavelet tree construction for the input 'he'. Each new line is after a *split&sort* and *pack* call. Each *pack* call is extracting the bold numbers. The *split&sort* call is sorting based on the bold character. 20

3.1. Diagram showing the threaded parts of domain decomposition. The example input has 32 characters, the calculation unit has 4 parallel units. n_x with $x \in 0, 1, 2, 3$, the x'th part of the input is meant. The picture shows, that the execution of λ_σ as well as the merging is parallelized. 22
3.2. Conceptional display of the Shun parallelization. Without displaying the creation of the bignode tree. 26

4.1. Comparison of tree construction for cell iterator and word iterator with a τ of 4. 31
4.2. Comparison of tree construction and matrix construction. Both with $\tau = 4$. As input, randomized content with $\log \sigma = 8$ is used. 32
4.3. The two plots show the internals of wavelet matrix construction, for $\log \sigma = 8$. On the left side for $\tau = 4$ on the right side $\tau = 2$. The prefix phase is the one creating the histogram, the bignode phase is building the bignode tree. And the pack&split phase transforms the bignode tree into the result object. Finally, this shows that the bignode building takes longer when there is a smaller τ. As input, randomized content with $\log \sigma = 8$ is used. 33
4.4. This plot shows the two different parallel matrix construction methods. The *a0* construction algorithm is the one from Shun, and *a1* is the domain decomposition. Running with 64 threads. As input, randomized content with $\log \sigma = 8$ is used. 34
4.5. Time spent for *splitting*, and *merging*. In domain decomposition compared to Shuns approach. The construction runs with $\tau = 4$ and 64 threads. . . 35
4.6. Weak and strong scaling experiments for the real world data. In weak scaling, 100M per thread are added. In strong scaling 6400MB are performed. 39
4.7. Weak scaling experiment, per thread 100MB are assigned. In each step 4 more threads are added. 44
4.8. Strong scaling experiment with 6,4GB of payload. With each step 4 more threads are added.

. . . . 44
4.9. Plot of memory usage divided by the payload size. The plot shows 6 cluster. Two clusters are from the PWM algorithm. Two from the normal *pext* algorithm and their domain decomposition versions. The last 2 clusters are from the Shun parallelization. For parallel constructions, 64 threads are used. 45
4.10. Plots showing the difference between the internal phases of the construction on a Intel E5 CPU on the left, and of the AMD CPU on the right. . . 45

List of Tables

4.1. All algorithms described in this work. All for $\tau \in 2, 4$. 29
4.2. Construction time with different block sizes τ in bits. Each algorithm name written as $wm_{\log \sigma}$. All evaluated with payload of the size 1000M. All times in *s*. $\tau > \log \sigma$ are not evaluated. The algorithm *wt0* is the word iterator, *wt1* is the cell iterator. 30
4.3. Runtime construction methods, parallelized using domain decomposition, for wavelet structures. The first two columns are for matrix construction, the left construction times for the merger described here, on the right those of the PWM repository. The second two columns are for tree construction. The right one for the matrix construction. As input, randomized content with $\log \sigma = 8$ is used. The columns called *Thesis Merger* is referring to the merger explained in 3.1. The columns called *PWM Merger* are referring to the merger of Dinklage et al. [2]. All values in *ms*. 33
4.4. These payloads are transformed into an alphabet without unused characters. This results in a more compact alphabet, where the $\log \sigma$ value has a direct impact on the depth of the levels that need to be walked. The script for transforming the input can be found in the repository of this work. From payload *CC* and *1000G*, only a 80GB prefix is used, as the original payload was too big for the available hardware. 36
4.5. Comparison of constructing wavelet tree times for real datasets. The *pext* algorithm is the cell iterator approach. Each result in *ms*. 37
4.6. Comparison of constructing wavelet matrix times for real datasets. Each result in *ms*. 37
4.7. Comparing construction times for parallel tree creation. Each result in *ms*. 38
4.8. Comparing construction times for parallel matrix creation.Each result in *ms*. 38
4.9. Sequential speedups of wavelet tree creation and the fastest PWM algorithm, with payloads with different $\log \sigma$. Each column is described as $s_{\lceil \log \sigma \rceil}$. With 64 threads. 41
4.10. Sequential speedups of wavelet matrix creation and the fastest PWM algorithm, with payloads with different $\log \sigma$. Each column is described as $s_{\lceil \log \sigma \rceil}$. With 64 threads. 41
4.11. Parallel speedups for wavelet tree creation with payloads with different $\log \sigma$. Each column is described as $s_{\lceil \log \sigma \rceil}$. With 64 threads. Further details can be received from A.1. 42
4.12. Parallel speedups for wavelet matrix creation with payloads with different $\log \sigma$. Each column is described as $s_{\lceil \log \sigma \rceil}$. With 64 threads. Further details can be received from A.1. 43

4.13. Different code sizes compared in lines. For *pext*, cell iteration for tree building is evaluated. The phase *flushing* is constructing the result buffer. For PWM algorithm this is the normal insertion of bits according to *pc*, *ps*, or *pc_ss*. For *pext*, *Shuns* algorithms this is the transformation from bignode tree to result. 46
4.14. Table showing the prognoses for the alternative algorithm approach described. The prognoses are calculated using Formula 4.1. The input values t_8, t_4, and t_2 are used from measuring the internals of the algorithms with $\tau = 8$, $\tau = 4$, and $\tau = 2$. All values in seconds 47

1. Introduction

The wavelet structure [13] is a space efficient lookup structure. The work of Dinklage et al. [2] also gives a survey, that gives further use cases in compression, computational geometry, and a helper to create the Burrows-Wheeler Transform. Further information can be found in the Introduction of Dinklage et al. It can be used to answer rank / select queries. The wavelet structure itself, is build from an input string. Each character of the input string is within the range $[0, \sigma)$. Therefore each character requires at least $\lceil \log \sigma \rceil$ bits to be stored. In practice, a single character is always stored in a multiple of 8-bit long character.
In the next Section 1.1, a brief overview of the current states is given, after that, the wavelet tree and matrix are explained in Section 1.2. Then, intrinistic functions, which will get used in this work are explained in Section 1.3. After that, basics for parallel algorithms are introduced 1.4. In the next Chapters 2 and 3 these basic blocks are used to introduce the algorithms.

1.1. Related Work

In this work a range of different algorithms are implemented. All of them do use assembler instructions used by Keneta in [9]. Additionally, those algorithms are compared to those from the work of Johannes Dinklage et al. [2]. They introduced a github [1] repository, this repository is collecting a wide range of algorithms. The algorithms pc, ps, pc_ss and the domain decomposition of those are used for comparison, as they are the fasted sequential construction algorithms, as well as the fasted parallel constructions. The algorithms are briefly explained in Section 4.4. The algorithms in PWM do construction in $O(n\log(\sigma))$. Munro et al. [4] and Babenko et al. [10] showed that construction is also possible in $O\left(n\log(\sigma)/\sqrt{\log(n)}\right)$ time, however, this was only shown theoretically. In 2018, Kaneta implemented those theoretical ideas in [9]. He used so called *pext* instructions to use RAM bit-wise parallism for the construction algorithms. He also implemented versions using *pshufl* instructions. However, the *pext* versions showed to be the fastest for tree constructions. For matrix construction *pext* and *pshufl* instructions both have been fast. However, in this work only *pext* versions have been used.
For parallel construction there are multiple construction algorithms in PWM [2]. A few do use customized parallelization strategies. Other versions in PWM do use domain decomposition. The fastest parallel algorithms are those utilizing domain decomposition. This also applies to other works from Fuentes-Sepúlveda et al. [5], and Labeit et al. [7]. Additionally to that, Shun [8] is proposing an alternative parallel algorithm that is not

[1] Github repository hosted at: `https://github.com/kurpicz/pwm`

using domain decomposition. This work also uses the ideas of Munro et al. [4] and Babenko et al. [10]. Shun proposes to use parallel integer sorting, described in [12] and [14], to run the first phase. The second phase is then parallelized individually.

1.2. Wavelet Tree and Wavelet Matrix

The wavelet tree and matrix structure is a perfectly balanced binary tree like object, with bit vectors in its nodes. For a given input sequence of length n, with a alphabet size of σ, the tree has $\lceil \log \sigma \rceil$ levels. Every edge between a child node and its parent has a associated 0 or 1. This association is later used to explain the construction, however, it is not stored. The exact meaning of these labels depends on if this is a wavelet tree or matrix. Every node it this wavelet structure has a level, the level is the number of edges between the node, and root, these levels are annotated as $i \in [0, \lceil \log \sigma \rceil)$. Each node in a level has an index x. In a level i there are 2^i different nodes. In the following two paragraphs, the relation between children and theire parent nodes are explained. The example memory representation in the next sections are representing the character sequence *hello world*. The bit representation of these characters are:

char	Bit representation							
h	0	1	1	0	1	0	0	0
e	0	1	1	0	0	1	0	1
l	0	1	1	0	1	1	0	0
l	0	1	1	0	1	1	0	0
o	0	1	1	0	1	1	1	1
	0	0	1	0	0	0	0	0
w	0	1	1	1	0	1	1	1
o	0	1	1	0	1	1	1	1
r	0	1	1	1	0	0	1	0
l	0	1	1	0	1	1	0	0
d	0	1	1	0	0	1	0	0

Wavelet Tree. For wavelet trees, with a parent node at index x the children indexes are $2x + 1$ or $2x$. Additionally, the cell at level i for character c can be calculated by obtaining the first j bits of the character c, these bits are then the index within the level. The resulting tree structure can be seen in Figure 1.3.
For building the wavelet tree, the first bit of each character is going to get copied into cell 0 at level 0. The second bit is added to the cell 0 of level 1, if the first bit is 0, or cell 1 if the first bit is 1. Recursively this means, that characters in level i from cell x are copied into cell $2x + 1$ at level $i + 1$ if the bit at position x of the character is 1, to cell $2x$ if the bit at the position is 0.
In Figure 1.3, the example wavelet tree of the input sequence "hello world" is shown.

Wavelet Matrix. In the following explanations, a function $reverse(b : byte, k : int) : byte$ is required. It takes the first k bits of the byte b and reverses their order, all bits $\leq k$ are not changed. This means, conceptionally: $reverse(abcdefgh_b, 4) = dcbaefgh_b$, and with

concrete numbers: $reverse(01110000_b, 4) = 11100000_b$. For each node at index x at level i the children are $reverse(x, i) \gg 1$ and $(reverse(x, i) \gg 1) \mid (1 \ll 7)$. With $\gg$ and $\ll$ symbolizing shifting operations. Building the wavelet matrix is the same as building the tree, just with the different formula for calculating the children. The difference in the built structure can be observed in the Figure 1.2 When constructing the wavelet matrix, the first child is used for insertion if the bit at position i is 0, the second child is used if the bit is 1. Additionally, the cell for character c in level i can be calculated by $reverse(c, i)$. Building the wavelet matrix is the same as the tree above. However, the children have a different positioning, as it can be seen in Figure 1.4 compared to Figure 1.3. This repositioning of the children has the advantage, that this lowers the amount of complexity required for building the wavelet matrix. This is further explained in the later chapter 2.2.2.

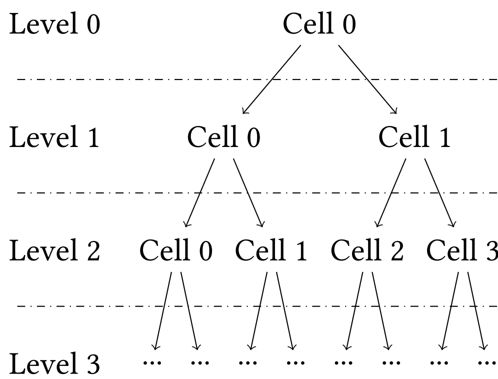


Figure 1.1.: Visualization of a wavelet tree for $\log \sigma = 4$. Each cell has always 2 children, and characters initially in the cell, must end up in one of its children. Additionally, the length of all cells of a level is always n.

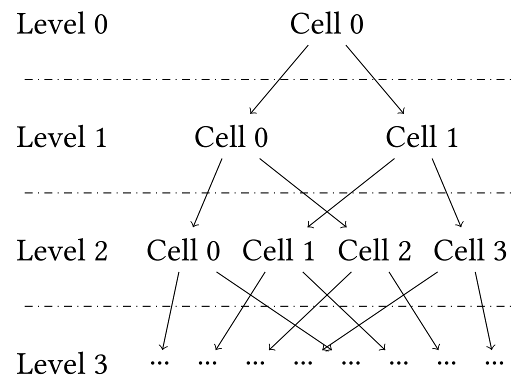


Figure 1.2.: Visualization of a wavelet matrix for $\sigma = 4$. Each cell has always 2 children, and characters initially in the cell, must end up in one of its children. The big difference to the tree is here, that the order of children cells is different.

Storage of Wavelet Tree and Matrix. Kaneta [9] is describing a few different ways of storing wavelet structures in memory. The easiest way of storing the wavelet structure is to have one memory object per cell, and each cell has two children cells, which are stored by pointer. The object itself then additionally has a vector of bits stored. This way of storing does not leverage the fact that the length of each line, hence the amount of bits per line, is known. However, when allocating cell per cell, the exact length is not known in the beginning, additionally, the vector of bits might have an overhead, as the smallest addressable entity is a 8 bit word, so there is a worst-case overhead of 7 bits per cell.
Following the idea of knowing the bit length per line, a single memory object for storing all bits of the entire structure arises. This object must be $n \log(\sigma)$ bit long. The bits of all cells of a level are forming the bits of a single line. The memory object can then be allocated as an array of 64 bit words. Figures 1.3 and 1.4 are showing how these layouts look like in memory.

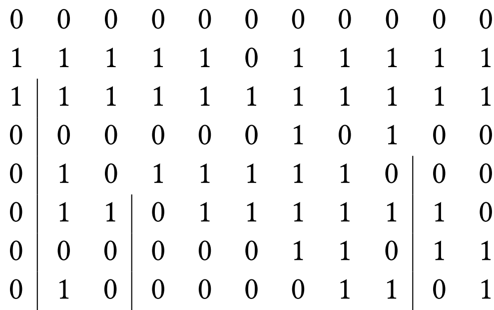


Figure 1.3.: The wavelet tree of the word *hello world.* The vertical lines are showing where each cell starts or ends. Not each line fills all available cells, hence the number of visible cells is not always the maximum that is possible.

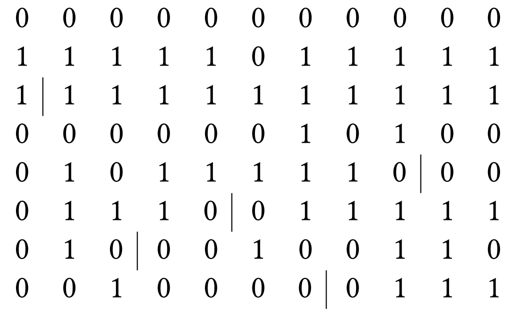


Figure 1.4.: The wavelet matrix of the word *hello world.* The vertical lines are separating the 0s cells from the 1s cells. If one cell is empty, no vertical line is added.

1.2.1. Rank / Select Queries on Wavelet Trees and Matrices

Three main queries that are performed on wavelet structures are access, rank, and select queries.

$access(pos : int) : char$**:** is returning the character on the position pos. For that, the wavelet structure is scanned through from the first level to the last. For each line iterated, a $position$ and $cell$ index is maintained. At level 0, the $cell$ 0 is used, and $position$ initially passed is used. In the iteration, $cell$ and $position$ is adjusted according to the bit retrieved at the $position$. If the bit is 1, the right child $cell$ is taken, and the position is updated to the number of 0's + the number of 1's until the $position$. If the bit is 0, the left child $cell$ is taken, the the position is updated to the number of 0's. In each iteration the retrieved bit is captured. The concatenation of these bits is the results from the access query.

$rank(a : char, pos : int) : int$**:** is returning the number of character a that is occurring until pos. For that, like in $access$, a $cell$ index and $position$ is maintained. At the first iteration $position$ is what got passed, and cell is 0. In each iteration the the bit from that level is taken out of a, if the bit is 1, the right child $cell$ is selected, and position is set the the number of 0's + the number of 1's until the current $position$. After the iterations the starting bit of the $cell$ is retrieved. The result of the function is then $position - cell_{starting}$.

$select(a : char, x : int) : int$**:** is returning the position of the n'th occurrence of a.
For this, a stack of min_i and max_i per level i is maintained. Initially for level 0 $min_0 = 0$ and $max_0 = n$. Then, in every line, min_{i+1} and max_{i+1} are set according to the ith bit in a. If this bit is 1, then $min_{i+1} = min_i + count_0(min_i, max_i)$ If this bit is 0, then $max_{i+1} = max_i - count_1(min_i, max_i)$ When this algorithm has reached the last level, then $cell_{min}$ and

$cell_{max}$ are the boundaries of the cell that is addressed by a. For the lowest level $min_{\log\sigma} + x$ is the position in the bit vector, that is describing the x'th occurrence of a in the string. Now, this position needs to be maintained up to level 0 in order to get the real position. For that, the iteration starts at the lowest level with $p = x$. For each level at i, the position for $i - 1$ can be calculated. If $min_i = min_{i-1}$ and $max_i = max_{i-1}$, then the position is not updated. If $min_i = min_{i-1}$ and $max_i \neq max_{i-1}$, then the bit at level $i - 1$ was 0, and the new position is the pth zero between min_{i-1} and max_{i-1}. If $min_i \neq min_{i-1}$ and $maxi = max_{i-1}$, then the bit at level $i - 1$ was 0, and the new position is the pth one between min_{i-1} and max_{i-1}. Finally, the the position is stored again into p. When this is executed until $i - 1 = 0$, then the position in the input string is found.

1.3. Additional Instructions

For exploiting word parallelism, Kaneta is using instructions like *pext* and *pshufl*. In this work, only the *pext* versions are implemented. Additionally *popcnt* is used for queries. The *pext* instruction is part of the *BMI2* (Bit Manipulation Instruction Set 2) instruction extension. It was first introduced in Haswell processors of Intel and the Excavator architecture from AMD.
The instruction *popcnt* is from the *ABM* (Adcanced Bit Manipulation) according to AMD, Intel considers it as part of SSE4.2.

pext Instruction. The *pext* instruction can be explained as $pext(a : word, mask : word) : word$ where word can be a 8/16/32/64 bit word. The bits of the result are extracted from a, where the mask bits are 1. Those extracted bits are then shifted to the right. The semantics of the instruction can also be explained with the Figure 1.5. In the following work, β is used to express the amount of bits processed as once with a single *pext* instruction.

popcnt Instruction. The *popcnt* instruction is counting the set bits in a passed word. The passed word can be a 8/16/32/64 bit word.

a	A	B	C	D	E	F	G	H
mask	0	0	1	1	0	1	1	1
result	0	0	0	C	D	F	G	H

Figure 1.5.: Explanation of the *pext* semantics. A/B/C/D/E/F/G/H are placeholders for bits.

1.4. Algorithem Parallelization

The algorithms in this work are implemented using C++. C++ has several threading frameworks. For example: boost, posix-threads, OpenMP. As this work is closely following PWM, the same framework, OpenMP, is used in this work. This framework allows to execute for loops and entire blocks in parallel. Additionally, the framework provides barriers for synchronization.
In general, parallelization can happen in two ways, either by splitting the problem into

parts, where each part gets solved in parallel, and then merged back into a single solution. Or by an individual algorithm, which does not use any sequential algorithm.

Domain Decomposition. The definition of domain decomposition, outside the scope of wavelet trees, from [3] is "Domain decomposition refers to partitioning of computational work among multiple processors by distributing the computational domain of a problem, in other words, data associated with the problem". This explains in a general way, that a sequential algorithm can be parallelized, under the assertion that multiple solutions can be merged into a single one. For implementing a solution like this, first the input is divided into multiple parts, which are solvable on their own. Then every part is computed with the sequential version of the algorithm. Finally, the solution of each computation must be merged into a single solution. The merging itself can then be parallelized again. However, not every problem can be split up into useful parts. As an example, calculating prefix sums can be split into different sub arrays. However, the merging back together part will take as much time as normally calculating the entire prefix sum, which renders the parallelization useless. However, a problem like creating a histogram over an int array is perfectly capable of being solved by domain decomposition, each thread can walk a part distinguished part of the input, the histogram of each thread can then be simply merged together by adding every cell of the histogram into the final one.
In the Section 3.1 this principle is applied to the problem of wavelet tree construction. The exact principal for merging solutions will be explained there.

Individual Parallelization. An alternative idea to domain decomposition is to run smaller parts of the algorithm in parallel. For that, different sequential operations can be broken up into different threads, just like in domain decomposition. The improvement here over the domain decomposition is, that by choosing the place for parallelization the time spend in merging the results can be minimized. To follow the example of histogram building, each thread could be assigned a range of elements where only that single thread is building the histogram for. This would result in the fact that no merging would be required. As each cell of the final histogram would only be written by a single thread. However each thread would have to scan the entire input.

Runtime Analysis of Parallel Algorithms. In order to analyze parallel algorithms the *work-time* (also called *work-depth*) model described in [6] as well as [11] is used. In this model the algorithm is analyzed by its *work* and by its *depth*. The analyzed work W is expressing how many operations are required in order to fully execute the algorithm. The depth D is expressing how long the longest single execution path from the beginning to the end is. The variable p is used to express the number of available parallel executing units.
Using the construction of the wavelet tree as an example. The input is split into two parts. Now the work W contains the amount of work required to construct the wavelet tree for the one half, as well as the other half, and finally the work of merging the solutions back together. The depth D is only the amount of work of one half, and the work required to merge.
After those two values have been analyzed, the parallelism of an algorithm can be expressed by W/D. Using Brent's scheduling theorem from [1], the running time can be bound by $W/p + D$.

1.5. Contribution

In this work, sequential and parallel wavelet construction algorithms leveraging *pext* instructions are implemented. These algorithms do have the runtime $O\left(n \log(\sigma)(\frac{1}{\tau} + \frac{\tau}{\beta})\right)$. Using *pext* instruction enables extracting a subset of bits out of a β bit long word. The basic idea of this paradigm was introduced by Munro et al. [4] and Babenko et al. [10]. Kaneta implemented these ideas for sequential algorithms and compared them to PWM in [9]. In this work, additionally parallel construction algorithms are performed and compared to PWM. For parallel construction, domain decomposition and parallelization according to Shun [8] are implemented and compared. It was learned, that the Shuns parallelization is too fine granular and therefore caused a too big overhead. More details for that are in Section 4.3. In the evaluation chapter, it is shown that the here implemented algorithms are faster than those in PWM, with the sequential versions for $\log \sigma > 3$, with parallel versions for $\sigma > 7$. The experiments compare the runtime as well as the memory consumption. While the runtime was able to be lower, the memory required increased. This is a result of the fact, that we need to bring the input into a memory layout where *pext* instructions can be used. The result from these experiments have been, that depending on $\log \sigma$ and n *pext* algorithms can be faster than PWM. Finally, in Section 4.2 it is evaluated which parameters result in the best runtime.

2. Sequential Algorithms

The construction algorithm described here follows the same pattern the algorithms from Munro et al. [4] and Babenko et al. [10] are following. They are splitting the construction into two phases, in the first, the *bignode tree* is constructed. In the second, the bignode tree is used to fill the final result. This second phase utilizes *pext* instructions, this leverages parallel processing of β bit long words. In Figure 2.1 this concept of the two phases is visualized. In this work here, this parallel processing of multiple bits is done with the so called *pext* instruction. Munro et al. and Babenko et al. are using look up tables for that. The first phase is then only there to create a memory layout that allows this instruction / table lookup to be used.
An addition to the paradigm from Munro et al. and Babenko et al. is the construction of a histogram in the first phase of the algorithm. This strategy is also used by Dinklage et al. [2] in the construction algorithms of PWM. Further more, this chapter explains two different ways to implement the wavelet tree. Over the process of this chapter, it is assumed, that $\tau < \log \sigma < n$.

Wavelet Structure Construction using two Phases

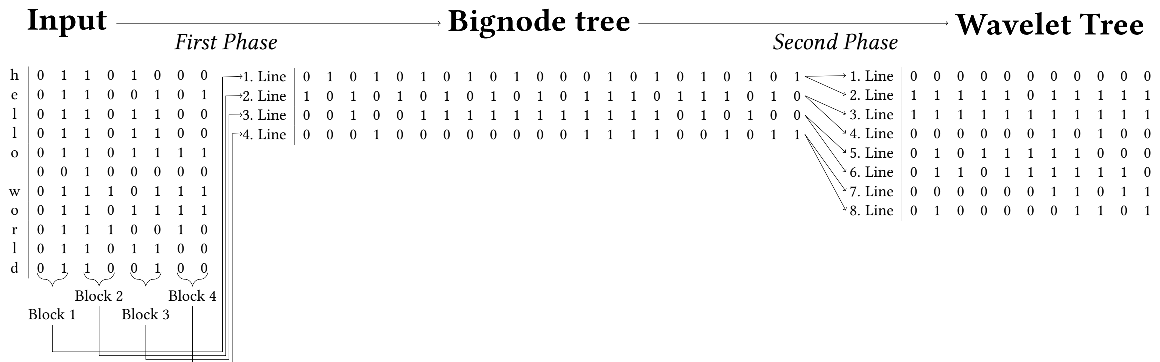


Figure 2.1.: Overview showing the relation between input, first phase, second phase, bignode tree, and resulting wavelet tree. The arrows of the first phase display that these values are represented in the associated line. The arrows of the second phase show in which lines the *pack* operation results are stored.

2.1. Bignode Tree Building

The bignode tree is a tree like structure. Each node of the tree has 2^τ children. Each child edge is associated with an $id \in [0, 2^\tau - 1)$. In this work, the level of the node, is the level

of the parent plus one. This results in $i \in [0, \left\lceil \frac{\log \sigma}{\tau} \right\rceil)$. The root node has the level 0. Each level has $2^{i \cdot \tau}$ nodes. Every node consists of τ bit long blocks. The sum of all blocks per node in level i is n. The node of a level i has an index x which is the bit wise concatenation of all edge ids, starting at root, going to the current node.
In order to get τ bit long blocks from the input character, the characters bit representation is split after τ bits. Which results in each character having $\left\lceil \frac{\log(\sigma)}{\tau} \right\rceil$ blocks. As an example, the bit representation of 'h' is $h := 01101000_b$ with $\log \sigma = 8$, its block representation is looking like:

$$h = \overbrace{01}^{b_0} \overbrace{10}^{b_1} \overbrace{10}^{b_2} \overbrace{00}^{b_3}$$

In case there are not enough bits in the original character, the missing ones are filled up with zeros.

Histogram. Before building the bignode tree, a histogram over the input is created. For the histogram, an additional binary tree like structure, which consists out of nodes is built. As its a binary tree, each level i has 2^i nodes, with $i \in [0, \lceil \log(\sigma) \rceil)$ levels. Each node at level i with position x is storing a counter. The node is counting how many characters are within $\left[\frac{\sigma}{i_{inverse}} x; \frac{\sigma}{i_{inverse}}(x+1)\right)$ with $i_{\text{inverse}} := \log(\sigma) - i$. When later building the bignode tree, only blocks of τ bits are stored, therefore the histogram is here storing the number of characters, multiplied with τ. Lastly, for each level i the prefix sum over each node at level i is calculated and stored.
The algorithms is build in 3 stages. First, the entire input is walked and the node in level $\lceil \log \sigma \rceil$ representing the character is increased. Due to the fact that this is a binary tree, the level at $\lceil \log \sigma \rceil$ has $\geq \sigma$ nodes. Therefore, each node is addressed by using the number representation of the character. After that, in the second stage, each node in level $i < \log \sigma$ must be build. Assuming level $i+1$ is already built, level i is built by calculating the counter of each node x by calculating the sum of both children. The indexes of the two children can be calculated with the formulas from the tree paragraph and matrix paragraph in the section 1.2. As the level at $\lceil \log \sigma \rceil$ is already filled from the first stage, the histogram is build by walking from the deepest level to the root at level 0. Finally, in the last stage the prefix sum of each node in a level is calculated, by walking over each node of a level and add the previous node to it. The first element of the prefix sum is always 0. This is required in practice as this histogram is used to store the state for bignode building.
After these stages, every last node in a level must be equal to n. For creating the histogram, the entire input must be scanned, which is $O(n)$, for building the histogram, $O(2^{\lceil \log \sigma \rceil})$ sums must be calculated and assigned. For the prefix sum, each node of the tree must be iterated, which is again in $O(\log \sigma)$. This creates the final runtime of:

$$O(n + \sigma) \tag{2.1}$$

Memory wise, only the counters must be stored, which requires words:

$$O(\sigma) \tag{2.2}$$

Bignode Tree building. Finally, the bignode tree is build. In the work of Babenko et al. [10] and Munro et al. [4], this step is described as the first phase. The bignode tree is build as an array of bit vectors. The array is $\left\lceil \frac{\log \sigma}{\tau} \right\rceil$ long. Each bit vector is $n\tau$ bits long. These bit vectors are filled by iterating over each character. As already described, a character can be split into blocks. A character is inserted by scanning over each block. For each block, the concatenation of the previous τ long blocks is used as an index x to find the correct histogram node at level i. For the first block, the index is 0. When building for a matrix instead of a tree, the index is bit wise reversed. The block is then written into the bit vector i of the bignode tree at the position stored in the histogram cell. The histogram cell is then incremented by τ. The result of such a bignode tree is displayed in Figure 2.2. This method is quite similar to the algorithm called *prefix counting* (*pc*) in [2]. The algorithm is briefly explained in Section 4.4.0.1. However, here always a block consisting of bits is inserted, not just a single bit. Lastly, the bignode tree is a compact memory structure, which is prepared in order to be able to use *pext* instructions later on.

1. Line	0	1	0	1	0	1	0	1	0	1	0	0	0	1	0	1	0	1	0	1	0	1
2. Line	1	0	1	0	1	0	1	0	1	0	1	0	1	1	1	0	1	1	1	0	1	0
3. Line	0	0	1	0	0	1	1	1	1	1	1	1	1	1	1	1	0	1	0	1	0	0
4. Line	0	0	0	1	0	0	0	0	0	0	0	0	1	1	1	1	0	0	1	0	1	1

Figure 2.2.: Example bignode tree for in the input sequence *hello world* and $\tau = 2$.

Runtime. In order to create a bignode tree, the entire input is scanned. For every character, $\left\lceil \frac{\log(\sigma)}{\tau} \right\rceil$ insertions are performed. This results in a runtime of:

$$O\left(n\left\lceil \frac{\log(\sigma)}{\tau} \right\rceil\right) \tag{2.3}$$

This runtime is equal to the first phase described by Babenko et al. [10].

Memory. The bignode tree stores $\lceil \log \sigma \rceil / \tau$ lines of $\tau \cdot n$ bits. Which results in a requirement of bits of:

$$O(\lceil \log(\sigma) \rceil\, n) \tag{2.4}$$

This is the same as the wavelet structure itself. In the following section, the bignode tree is used to create the wavelet tree and wavelet matrix.

2.2. Creating Wavelet Tree and Wavelet Matrix out of Bignode Tree

For the second phase, the bignode tree is used as an input. On each 64-bit word, the two operations *split & sort* and *pack* can be performed. First, we describe the two operations. Later, we describe how these operations are used to build trees and matrices. In this thesis,

we use β as an abstraction for 64-bit words.
For the two operations, there are two constants, ℓ and h, both are β bits long words. The constant ℓ is the concatenation of blocks where the lowest bit per block is set. The constant h is the concatenation of blocks where the highest bit per block is set. If $\beta > 0 \mod \tau$, then the last block is considered incomplete and gets ignored. All blocks are aligned at the MSB. Figure 2.3 is showing the value of ℓ and h for $\tau = 4$ and $\beta = 16$. Bitwise operations for left shifting, right shifting, exclusive or, and are denoted as $\ll$, $\gg$, $\oplus$, &. The next two paragraphs will explain two operations that are used over the process of constructing the matrix

constants	block 0				block 1				block 2				block 3			
ℓ	0	0	0	1	0	0	0	1	0	0	0	1	0	0	0	1
h	1	0	0	0	1	0	0	0	1	0	0	0	1	0	0	0

Figure 2.3.: Constant values ℓ and h for $\tau = 4$, which are used for *split&sort* and *pack* operations.

Split & Sort. The operation is annotated as $splitsort(block_position : int, value : word) : (word, word)$. With the operation, a single β bit word is virtually split into its blocks. These blocks are then sorted based on the bit at position $block_position$. All blocks with a 0 at $block_position$ are inserted into the first word. All blocks with a 1 at $block_position$ are inserted into the second word. The first word of the result is called L_0, the second L_1, both are also named L_x with $x \in \{0, 1\}$.
In order to calculate that result, first the so called $check_1$, $check_0$ values are calculated, they are also expressed as $check_x$ with $x \in \{0, 1\}$. After that $fill_0$ and $fill_1$ are calculated, they are as well expressed as $fill_x$ with $x \in \{0, 1\}$. Both are β bit long words. $fill_x$ is getting used as a mask for two *pext* calls, the result of this called is then the result of the entire call to this operation. The *pext* instruction has been explained in the introduction at Section 1.3.
The value $check_x$ consists of blocks, where the lowest bit of a block is 1, if the bit at $block_position$ is set to x, 0 otherwise. This can be calculated by:

$$check_1 := (value \gg block_position) \ \& \ \ell \tag{2.5}$$

And the opposite by:

$$check_0 := check_1 \oplus \ell \tag{2.6}$$

These two formulas do not change the order to the blocks yet. The values do have the relation, that $\ell = check_1 \ \& \ check_0$. This means, that each block from the $value$ is having its lowest bit set in either $check_1$ or $check_0$. So no block is going to be list by calculating a mask out of these two values.
In order to get a mask to select the correct blocks via $pext$, we need to set each bit in a block where the lowest bit is set. This can be calculated by:

$$fill_x := (h - check_x) \oplus h \tag{2.7}$$

This formula can be explained by looking at the binary representation of a subtraction of a single block. There are two possible ways for the subtraction either the lowest bit in the block is set as in Figure 2.5 or not, as in Figure 2.2. The result performed on an

a	1	0	0	0
b	0	0	0	0
a - b	1	0	0	0

Figure 2.4.: Subtraction result of a block with a MSB not being set

a	1	0	0	0
b	0	0	0	1
a - b	0	1	1	1

Figure 2.5.: Subtraction result of a block with a MSB being set

entire 64-bit word is resulting in the concatenation of these results. This can be proven by implicitly splitting the 64-bit word into numerical pieces, where each piece is one block. First, split $check_x$ into numerical blocks, where each block is $X_n \in \{0000_b, 0001_b\} = \{0, 1\}$:

$$check_x = \sum_{n=0}^{64/\tau} X_n 2^{n\tau}$$

Additionally, the same sum can be build for h:

$$h = \sum_{n=0}^{64/\tau} 2^{\tau-1} 2^{n\tau}$$

The subtraction $fill_x = h - check_x$ can now be rewritten as:

$$fill_x = \sum_{n=0}^{64/\tau} (2^{\tau-1} - X_n) 2^{n\tau}$$

$2^{\tau-1}$ is exactly the value of a in the examples. And X_n is exactly both possible cases of b. Lastly, the two subtractions do not cause any carryovers for the next block. Hence, $fill_x$ is just the bit concatenation of the blocks shown in Figure 2.2 and Figure 2.5.
As seen in the example figures, the entire block is set correctly, only the MSB is inverted. This can be repaired by performing xor with the constant h.
For both values of $fill_0$ and $fill_1$, no block positions have been changed yet. The two values do have the relation $\neg(fill_0 \oplus fill_1) = 0$. This shows, that for all blocks of the original *value*, no block is in two masks. Additionally, every block of the original *value* is going to be masked by one $fill_x$. Finally, the here calculated masks are used in two *pext* calls, which results in the L_0 and L_1. In order to receive how many blocks in each L_x, the bits of each mask $fill_x$ are counted, and divided by τ. This procedure can be executed by using the *popcnt* instructions. However, for the sake of readability, the number of valid blocks per word is not added to the function notation. As an example, Figure 2.6 shows the result and calculated variables of a *split&sort* call.

Pack. This work makes heavy use of bit blocks, that are concatenated into β bit long words. With the *pack* operation, a single bits out of every block is extracted. The extract

input	**0**	1	**1**	0	**1**	1	**0**	0
$check_0$	0	1	0	0	0	0	0	1
$check_1$	0	0	0	1	0	1	0	0
$fill_0$	1	1	0	0	0	0	1	1
$fill_1$	0	0	1	1	1	1	0	0

Result:

L_0	**0**	1	**0**	0	-	-	-	-
L_1	**1**	0	**1**	1	-	-	-	-

Figure 2.6.: Results of the *splitsort(0, input)* call. L_0 contains all blocks that have a 0 at block position 0. L_1 contains all blocks that have a 1 at block position 1. Each result has two blocks of result.

bits are then written next to each other in a single β long word. The function is called $pack(block_position : int, value : word) : word$. $block_position$ is describing to position of the bit within the block that is going to be extracted. $value$ is the β bit long word, which is the concatenation of bits. This is again done by first calculating a mask, calling *pext* with this mask, and returning the result. The mask is created, by left shifting the constant ℓ by $block_position$. The formula for that is:

$$mask := \ell \ll block_position \tag{2.8}$$

Lastly, the result of the function is simply the *pext* call with the calculated mask. In Figure 2.7 an example call, with the calculated mask is shown.

input	**0**	1	**1**	0	**1**	1	**0**	0
mask	**1**	1	**1**	0	**1**	1	**1**	0

Result: 0 1 1 0

Figure 2.7.: Results of a *pack(1, input)* call.

2.2.1. Wavelet Tree Construction

For creating the wavelet tree, the histogram and the bignode tree are passed as input. The goal of building the a wavelet tree is to fill $\log \sigma$ lines of n bits. This filled structure is then used later in query implementations for answering rank/select/access calls. The way how the wavelet tree is stored, is explained in the introduction at Section 1.2.
Briefly describing what needs to be done in order to construct the wavelet tree: With the bignode tree, a partially sorted tree is given as input. Each bignode tree level i is having the correct sorted order for line $i \cdot \tau$ in the result, this order only must be extracted into the result using *pack* operations. All lines in the result at $i \cdot \tau + \alpha$ with $0 < \alpha < \tau$ are not in the correct state yet. In order to correct this line, each block in this line must be sorted according to cells in the current line. This can be done, while assuming *split&sort* operations on a cell cover all blocks of a cell, by calling this operation on every cell of the line i. Every cell is then split into two child cells. These cells are then written next to each other, the exact order how children in a tree are ordered can be seen in Figure 1.1. Finally,

Histogram:

cell id	0	1	2	3	4
counter	0	4	6	10	16

Bignode line: 0 0 1 1 0 1 1 0 | 0 1 0 1 0 0 0 0

cell 0 (0 0 1 1), cell 1 (0 1), cell 2 (1 0 | 0 1), cell 3 (0 1 0 0 0 0)

Figure 2.8.: An example histogram line, and bignode line at $i = 2$. The bignode tree line is stored using $\beta = 8$. There are 4 cells in total, which are stored in 2 8-bit long words.

pack operations are used to extract the correctly sorted blocks.
In order to implement the creation, two additional lines with the sizes of bignode lines are allocated. In the following explanation, we make use of bignode lines associated with a level, written as j. For $j \equiv 0 \mod \tau$, j is the j/τ line of the originally passed bignode tree. For $j \not\equiv 0 \mod \tau$, every j has one of the two additional bignode lines assigned. This assignment ensures, that $j \neq (j+1)$. To build the tree, each level $i \in [0, \lceil \log \sigma \rceil)$ is iterated. In every level i, each β long word of the i bignode line is processed with the *pack* operation, and the results are written into the ith line of the result buffer. If $i \not\equiv \tau - 1 \mod \tau$, all cells $x \in [0, 2^i)$ are iterated. The exact starting position of each cell can be received from the histogram. For every β bit long word, that contains blocks from the cell x, *split&sort* must be executed. This exact task can be implemented in two different approaches, either by iterating cell by cell, or word by word. In the following paragraphs two implementations are explained, that are used to perform *split&sort* operations on all cells of a single line in the bignode tree. In order to explain the results better, we denote the results L_0 and L_1 of a single *split&sort* operation on a cell x, described as $r_{x,0} = L_0$ and $r_{x,1} = L_1$. The result of both implementations is a bignode tree line with the content of: $r_{0,0} \cdot r_{0,1} \cdot r_{1,0} \cdot r_{1,1} \cdot r_{i,0} \cdot r_{i,1} \ldots$, where $\cdot$ is interpreted as bit wise concatenation.

2.2.1.1. Using Cell Iteration.

Implementing this can be done by iterating over each cell x and its boundaries. This explanation will call i the index of the current line. The boundaries $[z, y)$ can be fetched from the histogram. From these boundaries the first, and last id of β bit long word can be calculated by $id_{first} = z/\beta$ and $id_{last} = y/\beta$. These words can then be iterated, and the *split&sort* operation on a single, β bit long word can be performed. Each L_0 and each L_1 can then be stored in the buffer of the next line. The position where to store L_0 or L_1 can be received from the histogram. To receive the exact position, the line i and cell $x * 2$ and cell $x * 2 + 1$ can be read. After L_0 and L_1 is written, the histogram cells containing the positions are updated accordingly. If a β bit long word is not entirely within a cell, a mask is used to mask the $fill_x$ variables in the *split&sort* operation, to only contain the blocks that are part of the cell.
As an example, Figure 2.8 is showing a line at $i = 2$ in the bignode tree, storing blocks within a $\beta = 8$. As it is in $i = 2$, there are 4 cells. Cell 0 is 2 blocks long, cell 1 is 1 block long, cell2 is 2 blocks long, and cell3 is 6 blocks long. The two words, which are used to

store the blocks are called $w0$ and $w1$. In order to process this line using the cell iterator, these 5 *split&sort* calls are done: w0 for cell 0, w0 for cell 1, w0 for cell 2, w1 for cell 2, w1 for cell 3. All of them have a mask applied in order to only process the blocks of the correct cell.

2.2.1.2. Using Word Iterator.

As shown in the previous paragraph, using the cell iterator results in multiple calls to *split&sort* with the same word. By iterating over each word, performing *split&sort* on it, and splitting L_0 and L_1 after each call. The split results are then stored in the $i + 1$ bignode tree line. When inserting L_0, we know that this is going to be in the position denoted as x_0, insertions of L_1 will go into position x_1. The splitting can be done, by checking how many blocks still need to be added to a cell, using the histogram. As explained in Paragraph histogram of Section 2.1, the cells of the histogram are a prefix sum, so the size of a cell in the wavelet tree can be calculated by subtracting the prefix sum at position x minus the position at $x - 1$. When the processing of the line starts, x_0 is zero and x_1 is one. When processing a word, the result L_0 is stored at the bit position in the histogram at x_0. When it is detected that the number of blocks in L_0 is bigger than the space left in the cell x_0, then x_0 is increased by 2, and the remaining blocks are stored in the new cell. If this cell is also not having enough space, the process of increasing x_0 and storing the next blocks there is repeated. The same process is repeated for storing the blocks of L_1. After each write of blocks, the histogram is updated accordingly.
Simulating this approach again for the example bignode line from Figure 2.8 will now only call *split&sort* two times, once for w0 and once for w1. However, comparing it to the cell iterator shows, that there is a logic overhead for handling the splits

2.2.2. Wavelet Matrix Construction

As described in Section 1.2, the difference between wavelet matrix and wavelet tree is the ordering of cells. The impact of that can be seen best in *split&sort* by rebuilding the word iterator approach for this. In order to reuse the word iterator approach for this, the formulas for calculating the children must be remodeled. In this case, x_0 is initialized with 0, x_1 is initialized with the value in the histogram cell at 2^{i-1}. This value is equal to the amount of zeros in the i'th level. Finally, x_0 and x_1 must be increased by one, if the cell is full. This means, that when the cell is full, the blocks will simply be added to the next word, which is equal to inserting the entire L_x to where the last write ended. This means, that the defined order of cells, removes the need to split the L_x results. This can also be seen when looking at the matrix visualization from the introduction in Figure 1.2.
For implementing the matrix creation, the same basic setup as for tree building is taken, the two additional bignode tree lines are allocated, and the same annotation, i, is used in order to express which bignode tree line is read / written to. Now, finally building the matrix, each level $i \in [0, \lceil \log \sigma \rceil)$ is iterated. In every level i, each β long word of the i bignode line is processed with the *pack* operation, and the results are written into the ith line of the result buffer. If $i \not\equiv \tau - 1 \mod \tau$, each β bit long word of i is iterated and split using *split&sort* operation. In order to write each L_0 and each L_1 two indexes

are maintained, called id_0 and id_1. id_0 is initialized with 0, and id_1 with the value of the histogram at level i and cell 2^{i-1}. For each *split&sort* operation, the resulting blocks in L_0 are written to the bit index stored in id_0. The blocks of L_1 to the bit index stored in id_1. After these writes, id_0 and id_1 are updated accordingly.

2.2.3. Influence of τ

As a short recap, the parameter τ is defining the number of bits per block in the bignode tree. Every line stores n blocks of bits. They are stored in β bit long words. Later on in the construction, these β bit long words are used for executing *pext* instructions. These instructions are used to leverage the possibility of parallel sorting using *split&sort* operations. In practice, the bit vectors are stored as 64-bit words, so $\lfloor \frac{64}{\tau} \rfloor$ blocks can be stored per word. Hence, a higher τ results in a lower number of blocks in a single β bit long word. Which overall results in a lower amount of parallelism leverages by *split&sort* operations. However, a higher τ also means, that the bignode tree creation does iterate over fewer blocks per character, and fewer memory writes in this phase are performed. Which lowers the expected runtime. This shows that τ must be evaluated while executing real runs, as the parameter can be used to move work between the first and second phase of the construction. The whole outcome of this effect is compared in the evaluation chapter 4.

2.2.4. Runtime

For transferring a bignode line to a result line, $O\left(\left\lceil \frac{n\tau}{\beta} \right\rceil\right)$ *split&sort* operations are required. This is ignoring that in tree creation, using the cell iterator approach, multiple *split&sort* operations are performed for the same word. Finally, $\log(\sigma)$ lines are getting transferred, therefore the complete transferring takes:

$$O\left(\left\lceil \frac{n\tau}{\beta} \right\rceil \log(\sigma)\right) \tag{2.9}$$

For reference, the algorithms from Munro et al. [4], and Babenko et al. [10] are taking $O(n \log \sigma\tau / \log n))$ in its second phase. As a short recap: Munro et al. as well as Babenko et al. do use pre computed tables. The key of the table has a bit length. Each possible key is in the table. The key can also be interpreted as a concatenation of blocks, where the value in the table is the same result as L_0 from *split&sort* operation explained here. The opposite operand L_1 can be calculated my inverting the key, and inverting the value. With the help of this table, *split&sort* operations can be implemented without using *pext* instructions.
The difference in runtime arises from the difference between the table, and β. While taking the approach of Munro et al. or Babenko et al., the length of the key can be freely chosen, as the table is created for the runtime of the algorithm. In order to minimize the runtime, the assertion $\log n \equiv 0 \mod \tau$ is taken, and a table with $\frac{\log n}{2\tau}$, entries is chosen. This way n must be divided into multiple parts of the length of the keys. Every part is then looked up and the results are stored. This means: $O(\frac{n\tau}{\log n})$ look ups are necessary in order to build a single line. Again, not accounting for looking up a single word multiple times, as it

would be required for the cell iterator approach.
When replacing the look up tables with *pext* instruction calls, the bit length of the keys is limited to what the hardware can provide. Therefore, β cannot be freely chosen in order to minimize the runtime. However, when assuming $\beta = \log \sigma$, for the sake of compatibility, the runtime of both algorithms is the same.
Summing up all the pieces in order to get the entire runtime of the construction: This is $O(n)$ for histogram building, $O(n \left\lceil \frac{\log(\sigma)}{\tau} \right\rceil)$ for bignode building, and $O(\left\lceil \frac{n\tau}{\beta} \right\rceil \log(\sigma))$ for creating the final result. This yields a final runtime as $O(n + n \left\lceil \frac{\log(\sigma)}{\tau} \right\rceil + \left\lceil \frac{n\tau}{\beta} \right\rceil \log(\sigma))$. In practice, the length of the input is ensured to be a multiple of β, therefore the ceil operation in the last phase can be removed. Additionally, the n part is dominated by the other terms, and is ignored therefore. Later on, in the evaluation, it will show that only τs dividing 8 evenly are considered. This finally means that the runtime reduces to:

$$O\left(n \log(\sigma)(\frac{1}{\tau} + \frac{\tau}{\beta})\right) \tag{2.10}$$

Again, the difference to Munro et al., and Babenko et al. is, that β cannot be chosen here in order to minimize the runtime, as this is bound for our approach to the size of hardware registers.
However, if $\beta = \frac{\log \sigma}{2\tau}$, and $\tau = \sqrt{\log \sigma}$, as it is in the work of Munro et al., and Babenko et al., the same simplifications can be done, and the result is: $O(n \log \sigma / \sqrt{\log n})$, which is the same runtime.

Memory usage. For memory usage, the construction requires the memory for storing the histogram, the bignode tree, the result, and the two swapping lines. For the histogram, $O(2^{\log \sigma})$ counters are required, each counter is some multiple of a byte. The bignode tree requires $O(\lceil \log \sigma \rceil n)$ bits. The result buffer requires again $O(\lceil \log \sigma \rceil n)$. And the two swapping buffers need $O(\tau \cdot n)$. Overall, this results in the overall memory usage of words:

$$O(\sigma + n \log(\sigma)) \tag{2.11}$$

2.3. Dynamic τ Adjustment

In this Section an alternative approach for constructing a wavelet structure is explained. In the work of Munro et al. [4] and Babenko et al. [10], a constant τ is assumed. In this new approach, τ is set dynamically per line.
The process of building the wavelet structure is split into the first phase, and the second phase. As a short recap, the first phase builds a memory layout that can be then be fast sorted and extracted into the result buffer. This memory layout consists of $\lceil \log \sigma \rceil$ so called packed lists. Each packed list consists of N τ bit long bit parts. The disadvantage of this is that the building of this memory layout also takes time. The gained performance however only happens during the second phase, as the first one is conceptional the same as *pc* in PWM. This phase could be replaced if the input is already in an usable memory layout.

The memory layout that is used as input of the bignode tree consists of n characters, where each character is a multiple of 8 bits. These characters are big enough, that $\log \sigma$ bits do fit into it. This means, that for example $\tau = 8$ and $\lceil \log \sigma \rceil = 8$ no first phase is required, as the layout itself is already correct. This is mostly caused by the fact that the amount of parallelization gained by a single *pext* instruction is not enough to be faster than the algorithms in the PWM repository.
The alternative approach to get rid of the first phase is to adjust the length of each τ per line. This can be achived by shrinking τ in the *split&sort* phase. The overall algorithm would then take the string as the input. First a histogram is build. The histogram is build in the same way described in Paragraph histogram in Section 2.1. Then, each line $0 \leq i < \lceil \log \sigma \rceil$ is iterated. In each line, for each cell $0 \leq x < 2^i$, the operations *split&sort* and *pack* are performed on the β bit long words. This can again be performed using the word iterator approach, or cell iterator approach. The insertion in the next line can be written with the indexes stored in the histogram. After each *split&sort* operation, L_0 as well as L_1 must be processed in order to lower the amount of bits per block by one. This can be done by masking the results L_x with a selector, which has only ones at those bits, that still need to be sorted. This selector can be calculated by: $(l \ll \tau - 1)\& \ldots \&(l \ll 1)\&l$. This means, that in each iteration, the memory allocated shrinks by n bits. Additionally, choosing τ on the fly for each line means that the value of ℓ and h must be calculated for each line. In Figure 2.9, the process of building a wavelet tree according to this approach is displayed. In each line, τ is getting decreased by one. Each character is exactly one block long. The evaluation will show in Section 4.2 that uneven τ are worsening the performance as they impose a logical overhead. Therefore $\tau \in \{8, 4, 2\}$ are valid candidates to choose from. Initially, $\tau = 8$ due to the nature that every character is having 8-bit. Now, after 4 *split&sort* calls, τ can be adjusted to 4, after 2 more *split&sort* calls, τ can be again adjusted to 2.
The idea of this algorithm appeared at the end of this work. For implementing this, nothing in the existing software structure could be used, as this always assumed a constant τ. Therefore, this approach was not implemented. However, the runtime is theoretically calculated in the evaluation chapter.

Runtime. For the entire construction, we would require $O(n + \sigma)$ memory reads and writes for building the histogram, as described in Paragraph 2.1. After that, in each line $i \in 1, \lceil \sigma \log \rceil$, $(in)/\beta$ *pext* instructions needs to be executed, as the amount of *pext* instruction decreases with every line, where the bignode lines do get smaller. This results in the overall runtime:

$$O(\lceil \log \sigma \rceil \frac{n}{\beta}) \tag{2.12}$$

Memory Consumption. Implementing this requires the histogram, two swapping lines, and the input buffer. The histogram requires $O(\sigma)$ counters of memory, as described in 2.2. For the two swapping lines, and the input buffer $O(n)$ bytes are required. Which results overall in words of:

$$O(n + \sigma) \tag{2.13}$$

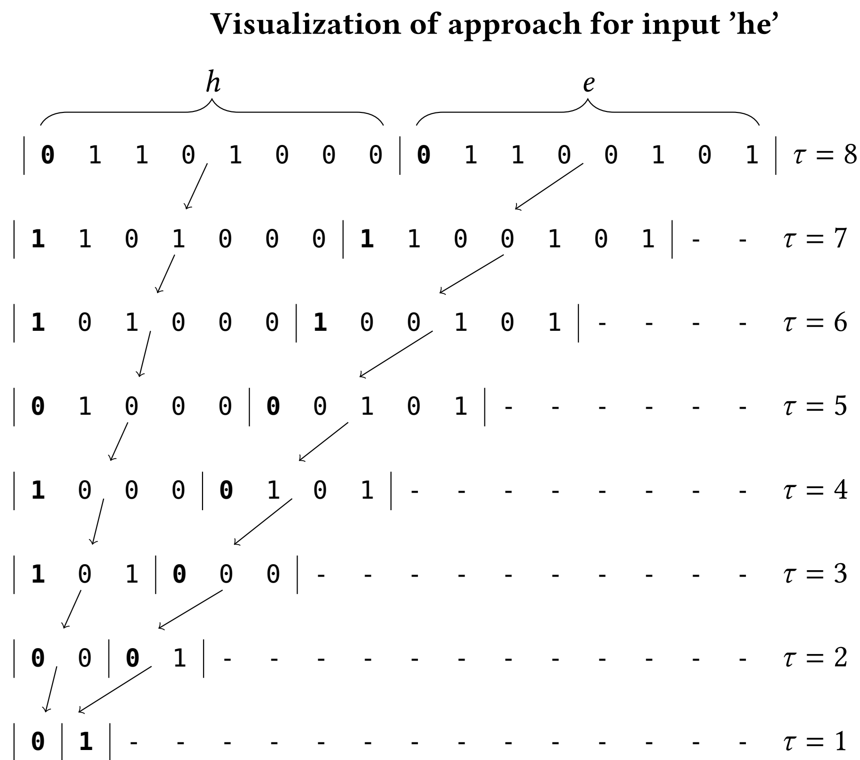


Figure 2.9.: Wavelet tree construction for the input 'he'. Each new line is after a *split&sort* and *pack* call. Each *pack* call is extracting the bold numbers. The *split&sort* call is sorting based on the bold character.

3. Parallel Algorithms

In this chapter, two ways of parallelization are introduced and explained. First domain decomposition with a explanation how multiple wavelet trees can be merged into a single one. After that Shuns [8] approach. Later in the evaluation, the merger explained here will be compared to the one of the PWM repository. When speaking of parallel algorithms, it is meant, that the algorithm has parts that are executed in parallel. The parallel execution happens on p different units. So the same algorithm is executed p times in parallel. These units are called threads. All units executing instructions in parallel are accessing the same memory. Over the process of this work, $0 \leq t < p$ is used as a numerical identifier for each thread. The same assumption as for the sequential algorithm are taken here: $\tau < \log \sigma < n$.

3.1. Domain Decomposition

As already described in the Introduction in Chapter 1, when executing an algorithm in parallel, using domain decomposition, the splitting of the input needs to be defined, as well as the merging of the results. Each part of the split input is then transformed using the sequential version of the algorithm, this execution will be denoted as $\lambda_\sigma(n)$. After that, the result of each execution t, denoted as $r_t := \lambda_\sigma(n)$ is merged back together.
For splitting, the input is just split into t parts. The length of the different parts are ensured to be evenly dividable by 64. Then the results are created using the matrix creation, tree creation with cell iteration as well as the word iteration. Lastly, t different wavelet structures need to be merged into a single one. In the following, $x_t \in [0, 2^{i_t})$ in level $i_t \in [0, \lceil \log \sigma \rceil)$, are defined to address the cell or level of the wavelet structure resulting from the parallel execution t. In order to merge them back together, each cell $x \in [0, 2^{i_t})$ in level $i \in [0, \lceil \log \sigma \rceil)$ of the resulting wavelet structure is iterated. For each x, all bits of x_t with $0 \leq t < p$ are written next to each other.
For implementing this, the splitting can be realized without copying, by only passing pointers to the string, with the length of the segment. The merging back together, is also parallelized. For that, first the histogram of all p wavelet structures have been merged into a single one. This can be done by simply summing up every counter of them. After that, each thread t is assigned a range of bits with a starting bit and a length. Each thread is copying exactly those bits described by the start bit and the length to the result buffer, for all lines $i \in [0, \lceil \log \sigma \rceil)$. The length is therefore a multiple of 64-bits, this way it is ensured that each thread can copy entire 64-bits words, if the cells are bigger than that. The threads can find the position where to start copying by scanning through all histograms of the threads and starting when the sum of previously copied bits is getting equal to the starting bit. Then they just iterate through all threads, and copy the number of cells in order to fill the length of bits. This algorithm is conceptional showed in Figure 3.1. This method of

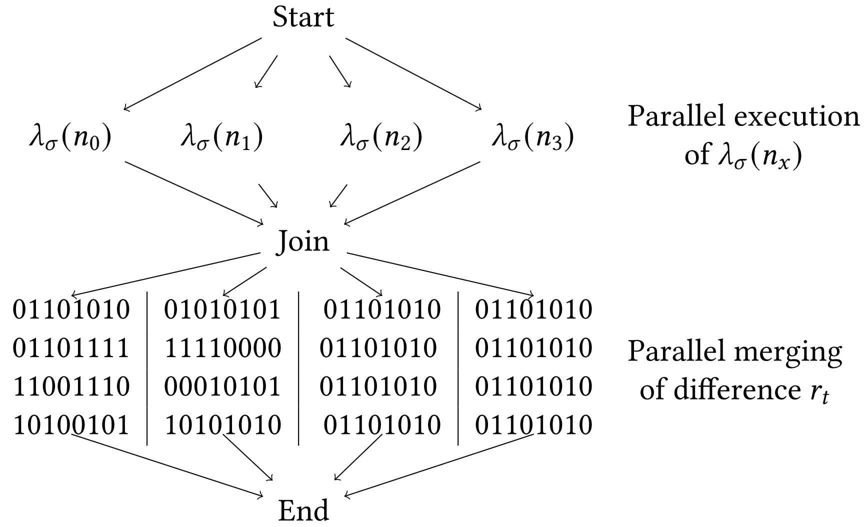


Figure 3.1.: Diagram showing the threaded parts of domain decomposition. The example input has 32 characters, the calculation unit has 4 parallel units. n_x with $x \in 0, 1, 2, 3$, the x'th part of the input is meant. The picture shows, that the execution of λ_σ as well as the merging is parallelized.

merging already used and implemented in PWM. The fastest three algorithms from PWM do use domain decomposition.

Runtime. As explained, the domain decomposition uses a sequential algorithm. Therefore, the runtime depends on the runtime of the sequential algorithm, $O(\lambda_\sigma(n/p))$,. Additional parameters like τ or β or not noted, however, the parameter for each execution would be the same.
The first phase of splitting can be done in constant time. After that, the execution of $p\lambda_\sigma(n)$ is needed. Finally, the results need to be merged together. For that, each thread is copying the bits relevant to him. The actual position where to start copying can be calculated from the histogram of each thread. This requires $O(p)$ time, after that, the copying takes $\frac{n}{p}\log\sigma$ operations which results in $O(p + \frac{n}{p}\log\sigma)$ operations per thread when merging. Summing up the operations required for wavelet structure construction, and the merging of the wavelet structure, requires work:

$$O\left(p\lambda_\sigma(n/p) + p(p + \frac{n}{p}\log(\sigma))\right) \tag{3.1}$$

For depth, the operations on one sequential construction plus the operations of one thread merging contents is required:

$$O\left(\lambda_\sigma(n/p) + p + \frac{n}{p}\log(\sigma)\right) \tag{3.2}$$

With the runtime from wavelet construction in Equation 2.10, this results in work:

$$O\left(n\log\sigma(\frac{1}{\tau} + \frac{\tau}{\beta}) + p^2 + n\log\sigma\right) \tag{3.3}$$

And depth:

$$O\left(\frac{n}{p}\log\sigma(\frac{1}{\tau}+\frac{\tau}{\beta})+p+\frac{n}{p}\log\sigma)\right) \tag{3.4}$$

Memory usage. The sequential implementation of each thread requires $O(2^{\log(\sigma)}+\log(\sigma)(\frac{n}{t}))$ bytes of memory, which is explained for Equation 2.11. This is required p times. Additionally, the buffer for the end result requires $O(n\log\sigma)$ bits of memory. Finally, the memory requirements in words are:

$$O(\sigma+\log(\sigma)n) \tag{3.5}$$

The required memory for the domain decomposition scales the same as the sequential version.

3.2. Shun Parallelization

Shun [8] introduced a parallel construction algorithm for wavelet trees. The algorithm is similar to the paradigm followed here. First, a bignode tree is built, then the bignode tree is transferred into the result buffer.

Bignode Tree building by Shun. Shun proposed to build the bignode tree using stable sort algorithms. For each line i in the bignode tree, all characters in the input array are sorted according to the key. The key, is the prefix of blocks. The prefix of blocks, are the $\tau\cdot i$ bits before the current block. So for line $i=2$ the character h:

$$h=\underbrace{\overbrace{01}^{b_0}\;\overbrace{10}^{b_1}}_{\text{prefix}}\;\overbrace{10}^{b_2}\;\overbrace{00}^{b_3}$$

This example shows, for $i=0$, the prefix is empty, which results in no sorting work. For all lines $0<i<\lceil\log\sigma\rceil$, all lines are sorted. When a line is completely sorted, the current blocks are extracted from the character and appended to the bignode tree line. After that, the next line can be sorted. In order to build the bignode tree for matrix instead of the tree, the key can be calculated, by reversing the bits of the key. For sorting algorithms, Shun proposed two algorithms:

1. An algorithm with work of $O(n\log(\log(n)))$ and a depth of $O(\log n)$ which would make the whole building work $O(n\log(\log(n))\lceil\log(\sigma)/\tau\rceil)$ and depth $O(\log n\lceil\log\sigma/\tau\rceil)$. The algorithm is explained in [12].

2. An algorithm with work of $O(n/\epsilon)$ and depth of $O(n^{\epsilon}/\epsilon)$ with $0<\epsilon<1$. Which results in a total work of $O\left((n/\epsilon)\left\lceil\frac{\log\sigma}{\tau}\right\rceil\right)$ and $O\left((n^{\epsilon}/\epsilon)\left\lceil\frac{\log\sigma}{\tau}\right\rceil\right)$ depth. The algorithm is explained in [14].

The difference between the two algorithms is that 2) is work efficient, 1) is not.

Bignode Tree building with Domain Decomposition. Another possibility for bignode building is taking the sequential algorithm presented in Section 2.1, and run it in parallel with domain decomposition. This can be done following the same pattern as the domain decomposition of the entire wavelet structure construction. First we define the splitting of the input. Followed by the merging back together p different bignode trees. The input string is split into p evenly sized segments. After that, the bignode trees are build for all parts. Lastly, these different bignode trees need to be merged back together. They can be merged in the same way as the wavelet structure itself. We denote x_t as the index of the cell in line i_t of wavelet structure t. Now, for every cell x in line i of the final bignode tree is iterate, for these cells, every cell x_t from thread $0 < t < p$ is getting written next to each other. In order to implement this, each thread is building the histogram as described in Paragraph histogram in Section 2.1. After that, each bignode tree is build in its thread t. Finally, the bignode tree cells are walked and merged, as described above. The histogram building for all threads, takes $O\left(p * 2^{\frac{\log \sigma}{\tau}}\right)$. Building the bignode trees is within: $O\left(n\tau \left\lceil \frac{\log(\sigma)}{\tau} \right\rceil\right)$. The merging takes: $O\left(n\tau \left\lceil \frac{\log \sigma}{\tau} \right\rceil\right)$ write operations. The results in a total runtime of:

$$O\left(n\tau \left\lceil \frac{\log \sigma}{\tau} \right\rceil + p * 2^{\frac{\log \sigma}{\tau}}\right) \tag{3.6}$$

and depth:

$$O\left(\frac{n\tau}{p} \left\lceil \frac{\log \sigma}{\tau} \right\rceil + p * 2^{\frac{\log \sigma}{\tau}}\right) \tag{3.7}$$

This shows that the algorithms proposed by Shun are theoretically scaling better. However, implementing Shuns approach brought up the following problems:

Using Arbitrary Integer Sorting for Bignode Building. A implementation problem of this approach is, the building of the actual bignode lines. Babenko et al., Munro et al., and Kaneta do call this list of blocks *packed lists*. They are required in order to extract value either due to *pext* instructions, or key value table look up. In order to build them while integer sorting, these packed lists need to be written. In order to do so, the algorithm must be able to synchronize/organize writes to RAM words in a way, that no other thread is writing to the same β bit long word, as that would overwrite previous writes. If this is not the case, another parallel walk over the entire sorted array must be done, extracting the required τ bits.
Additionally, there are two different paradigms that either a bignode tree or the entire structure can be created.

1. Iterating over each character, and insert every block into the correct position in the line. This way, previous knowledge over where each cell starts must be in place, this is realized using the histogram.

2. Iterating over each line, walking over each character, inserting the required block from the character, storing the block, as well as the entire character in the next line. Storing the character is required to maintain its internal order, for the next sorting iteration.

Both of these solution have theoretically the same number of insertions. However, paradigm 1 requires to have knowledge over where to insert a specific part of a character beforehand. And paradigm 2 requires to store a buffer with relative sorting after each line. Both paradigms have been implement in PWM. Paradigm 1 is followed by *wt_naive*, paradigm 2 is followed by *wt_pc* and others. What shows based on the construction times from *wt_naive* compared to *wt_pc* is that paradigm 2 is a lot faster, since writing to the additional buffer is way too slow. The algorithms referred here are explained in Section 4.4.0.1.
Summing up: Using integer sorting for bignode building requires more as twice as much memory writes, compared to bignode building described in Section 2.1. Therefore, in this work, the domain decomposition of bignode tree construction is used and evaluated, instead of using integer sorting.

Wavelet Tree and Wavelet Matrix creation. After the creation of the bignode tree, each level is then again iterated. In the work of Shun, tables are used where parts of bignode tree is the key, and the value is the result of the block sorting. In the work of Kaneta [9] it is shown, that these tables can be replaced using *pext* instructions. In the following implementation explanation, the same notation j as in Section 2.2.1 is used. The algorithm proposed by Shun iterates over each line $i \in [0, \lceil \log \sigma \rceil]$. In each line, the entire bignode tree line is split into p evenly sized chunks. These chunks are denoted as $c_{i,t}$. For every chunk in parallel, the splitting as well as extracting is performed, this processing is denoted as $P(c_{i,t})$. This can be implemented by iterating each cell in the chunk, and writing the L_0 and L_1 results of *split&sort* operation into the according buffer. The result of each operation is stored in a buffer allocated exclusively for each chunk. After the operations on each chunk has been finished. First, the results of the result line i are written, by walking through the chunks and writing the pack results of each chunk next to each other into the result buffer. Secondly, the results of the *split&sort* must be written into $j(i+1)$. This is done by iterating through all chunks, and write all contents to the cell where they belong to. The position for that write can be received from the histogram. After each write, the histogram is updated. For the matrix, the chunk will only have two result cells, therefore only to two cells can be written, the positions for that are again known from the histogram, which is updated after each write. After this process is repeated $\lceil \log \sigma \rceil$ times, the entire result buffer is filled. This algorithm is visualized in Figure 3.2, every parallel execution is displayed as a line in the image each arrow symbolizes a thread. The major difference to domain decomposition is how many different groups of threads are started and joined back together. In the domain decomposition only 2 groups are ran. The Shun [8] parallelization runs the 1 group from the bignode creation plus $\lceil \log \sigma \rceil$ groups for the transformation to the result buffer. This solution has its difference from the domain decomposition that the parallel execution is happening finer graded. In Domain Decomposition the entire structure construction is running in a single thread. Here, the threading is happening for bignode tree construction and each line of the flushing process.

Runtime. In each line $O(n)$ work has to be performed, with a depth of $O(n/p)$. This results in total of $O(n \log \sigma)$ work for constructing the wavelet structure with a given bignode tree, and $O(n/p \log \sigma)$ depth.

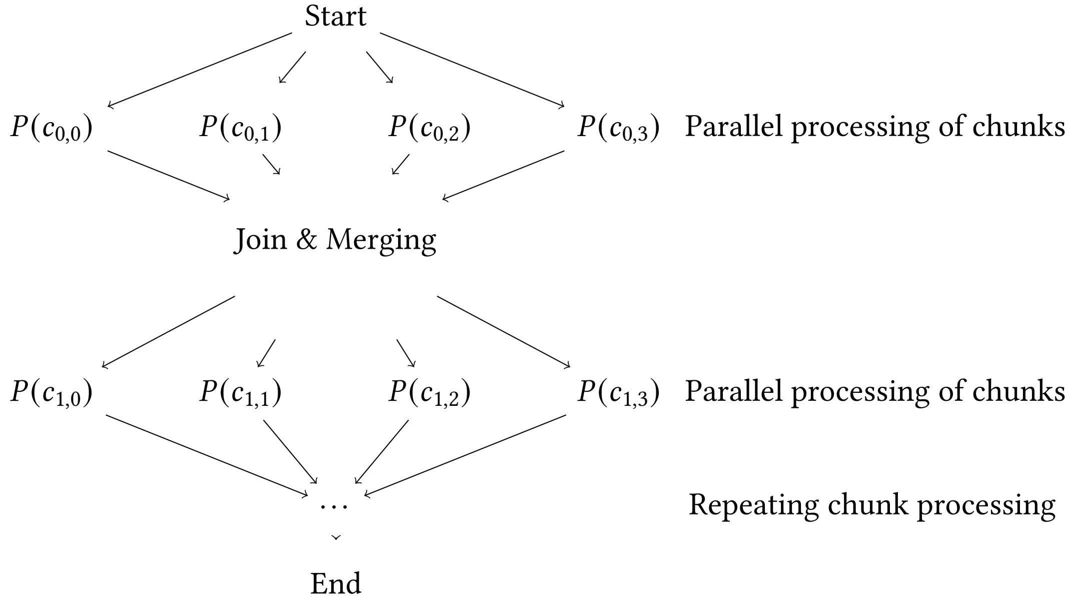


Figure 3.2.: Conceptional display of the Shun parallelization. Without displaying the creation of the bignode tree.

For the entire algorithm, the bignode building with work of Equations 3.6 and depth of Equation 3.7 is performed. Due to that, this results in work:

$$O\left(n\left\lceil\frac{\log\sigma}{\tau}\right\rceil + p * 2^{\frac{\log\sigma}{\tau}} + n\log\sigma\right) \tag{3.8}$$

and depth:

$$O\left(\frac{n}{p}\left\lceil\frac{\log\sigma}{\tau}\right\rceil + p * 2^{\frac{\log\sigma}{\tau}} + n/p\log\sigma\right) \tag{3.9}$$

Memory usage. The general memory setup is the same as in the normal construction. Two buffers for writing the *split&sort* results. The bignode tree in its normal form. Additionally to that, each chunk needs to have a buffer, where *split&sort* results are written before they are getting merged.
The size of this buffer depends on if the structure is either a tree or a matrix. For splitting a single chunk in trees, 2^i possible cells can be filled. In worst case, only one cell will be filled with the entire bits from the chunk. Therefore the buffer must be $2^i * n_c$ where i is the current depth that is processed, and n_c the length of the chunk. For matrices, only 2 possible cells can be filled. Therefore the size of this buffer needs to be $2 * n_c$.
This results in a final memory usage for matrices in words of:

$$O(\sigma + n\log\sigma + 2n) \tag{3.10}$$

And for trees:

$$O(\sigma + n \log \sigma + 2^{(\log \sigma) n}) \tag{3.11}$$

4. Evaluation

In this chapter, the different algorithms described earlier are evaluated. All the algorithms are listed in Table 4.1. First, the evaluation shows which wavelet structure construction

Algorithm title	Description
$wm_pextb\tau a0$	normal matrix creation 2.2.2
$wm_p_pextb\tau a0$	parallel, Shun 3.2
$wm_p_pextb\tau a1$	parallel, domain decomposition 3.1
$wm_p_pextb\tau a2$	parallel, PWM domain decomposition Dinklage et al. [2].
$wt_pextb\tau a0$	Word iterator 2.2.1.2
$wt_pextb\tau a1$	Cell iterator 2.2.1.1
$wt_p_pextb\tau a0$	Word iterator 2.2.1.2, parallel, domain decomposition 3.1.
$wt_p_pextb\tau a1$	Cell iterator 2.2.1.1, parallel, domain decomposition 3.1.
$wt_p_pextb\tau a2$	Word iterator 2.2.1.2, parallel, domain decomposition from Dinklage et al. [2].
$wt_p_pextb\tau a3$	Cell iterator 2.2.1.1, parallel, domain decomposition from Dinklage et al. [2].

Table 4.1.: All algorithms described in this work. All for $\tau \in 2, 4$.

method is the fastest for its structure type. For that, cell iterator and word iterator for wavelet tree construction are compared. For parallel construction of wavelet matrix, Shuns approach is compared with domain decomposition. Additionally, for the parallel construction the merger from PWM is compared with the merger presented in this thesis. Lastly, the impact of τ is measured. For the evaluation the algorithms have been executed on a Intel(R) Xeon(R) CPU E5-2683 v4 @ 2.10GHz CPU. The CPU has 2 sockets, with 16 cores per socket, and 2 threads per socket, resulting in 64 parallel execution units.

Experiment Payloads. For the experiment, two different types of payloads are used. Randomly generated payloads, where each payload with a certain size is created by adding random characters from */dev/random* into a file.
The second type of payloads are the first five PWM [2]. They are known as XML, DNA, ENG, PROT, CC, 1000G, and SRC. The details of them can be taken from Table 4.4. The exact histograms of these payloads can be seen in A.2. XML consists of multiple XML documents. DNA contains sequenced DNA data with 16 different characters. ENG is a selection of english texts. PROT is a selection of stored proteins. SRC is a selection of c source code. The biggest payloads are CC and 1000G. CC is a collection of wikipedia articles. 1000G is a set of DNA elements from the 1000 Genomes Project.

4.1. Sequential Runtime of Basic Settings

In this section the different versions of algorithms are compared. Additionally, the impact of the settings β and τ are compared. As a short recap: The algorithms first build the bignode tree, which consists of τ bit long blocks per line. In the second phase, these bignode tree lines are processed with *pext* instructions. These instructions can process β bits at once.

Different τ values. All three algorithms presented in this work, can be executed with different block sizes. The different construction times can be compared in the Table 4.2. It can be observed, that for $\lceil \log \sigma \rceil = 8$ and $\tau = 4$ all algorithms are the fastest. For other $\log \sigma \neq 8$, $\tau = 2$ are the fastest. Additionally, uneven bits are slower than even bits. This can be explained with the overhead uneven τs cause. First, bignode lines cannot be used entirely, as for $\beta \equiv 0 \mod \tau$, the last $\lfloor \beta/\tau \rfloor$'th block of τ bits cannot be used. Furthermore, when packing bignode lines into the result buffer, additional conditional jumps are required, as not every τ pack call is completing an entire 64 bit word in the buffer. This finally results in more instructions being executed for packing a single result line.

τ	wm_8	$wt0_8$	$wt1_8$	wm_6	$wt0_6$	$wt1_6$	wm_4	$wt0_4$	$wt1_4$	wm_2	$wt0_2$	$wt1_2$
2	19.1	19.2	18.6	**14.1**	**13.7**	**13.2**	**10.3**	**9.9**	**9.6**	**6.8**	**6.8**	**6.8**
3	24.2	24.4	23.4	16.3	17.1	16.4	14.3	14.5	14.2			
4	**17.5**	**19.1**	**16.9**	14.9	15.6	14.3	10.8	11.5	10.6			
5	23.2	26.3	23.7	18.5	20.0	18.5						
6	27.9	31.6	27.9	17.5	21.8	19.3						
7	32.5	36.5	33.9									
8	26.8	31.7	27.9									

Table 4.2.: Construction time with different block sizes τ in bits. Each algorithm name written as $wm_{\log \sigma}$. All evaluated with payload of the size 1000M. All times in *s*. $\tau > \log \sigma$ are not evaluated. The algorithm *wt0* is the word iterator, *wt1* is the cell iterator.

Comparing Cell Iterator and Word Iterator. In Figure 4.1 the difference between cell iterator and word iterator is shown. The cell iterator, which is displayed red, is faster than the word iterator, which is blue. When comparing the implementation details of both implementations, the word iterator requires more code for performing writes for a single β bit long word compared to the cell iterator, as the result of a single word can be part of multiple cells. In order to implement that, a while loop is executed which scans over how many bits are left from the last cell, and either the whole word is added to this cell, or a smaller amount. The downside of this is, that before adding bits to a cell, the while loop is entered, and the current state of the cell is checked.
These problems are solved in the cell iterator, as the cell itself is always known by definition, as all cells are iterated one by one. Additionally, the current state of the cell can also just be kept within the iteration, which at least replaces the fetching of this state from within

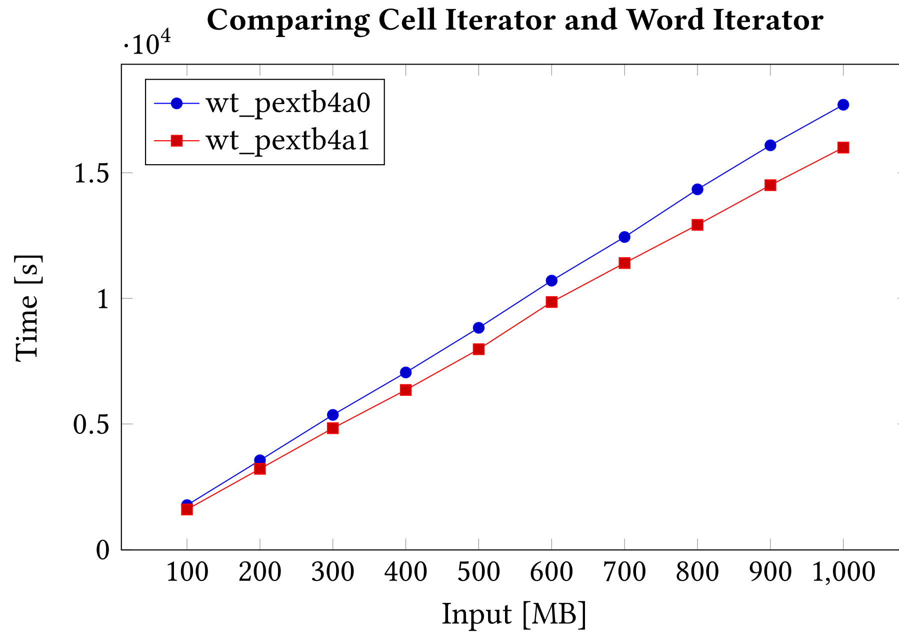


Figure 4.1.: Comparison of tree construction for cell iterator and word iterator with a τ of 4.

the history. Additionally, if the cell is full must be checked in the last word, and that can be easily calculated using the histogram and keeping track of how many bits are already processed in this cell.
The only positive argument for the word iterator approach is, that it minimizes the amount of pext calls. As the cell iterator would perform the same operations twice on the words where two or more cells are part of. However, assuming the input size n grows, the number of duplicated *pext* calls can only be $2^{\lceil \log \sigma \rceil}$, which invalidates the initial approach.

Comparing Wavelet Tree with Wavelet Matrix Construction. Comparing the tree and matrix construction, one can see in Figure 4.2 that they have a almost identical runtime. Only in upper payload sizes tree construction is slightly faster.
Comparing again the details of tree and matrix: First, the tree needs to handle all cells in a single line. Matrices do not need that. However, matrices do need to reverse the cell address in order to get the real address. This reversing is required when building the histogram, as well as when the bignode tree is build. Not for transferring the bignode tree to the result.
Lastly, the overhead of reversing the addresses cancel out most of the overhead caused by handling every cell.

Internals of Sequential Algorithm. As described in Section 2, the algorithm works in two phases: bignode tree building, filling the result buffer. Bignode tree building is for measurement purposes split into the bignode building and histogram building. In Figure 4.3 the 3 different phases can be seen for $\tau = 4$ on the left side, and $\tau = 2$ on the right side. The big difference is that the bignode tree building phase takes longer than before.
Additionally, the two plots can be used to verify if the scaling of the different τ values do

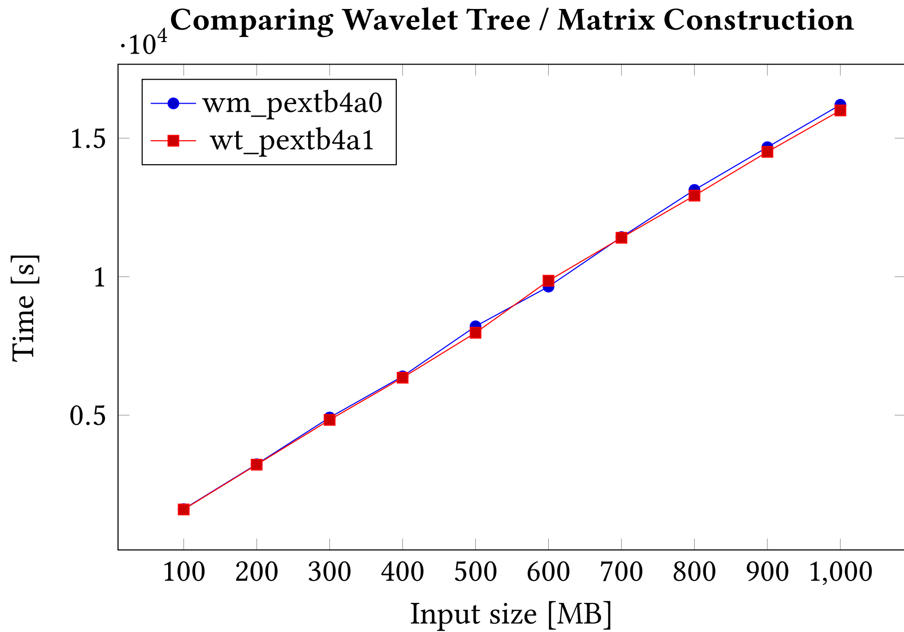


Figure 4.2.: Comparison of tree construction and matrix construction. Both with $\tau = 4$. As input, randomized content with $\log \sigma = 8$ is used.

show the impacts visible from the Runtime O-Notations. For the histogram building, this shows that variations in the τ variable does not have an impact on the runtime. For the bignode tree building phase, this shows that a bigger τ reduces the amount of time spent in building. Finally, the time spent in flushing the bignode tree to the result is growing with a bigger τ. The experiment itself shows that the flushing does scale better than the bignode tree building. This can also be seen when comparing the runtime O-Notation of bignode building in Section 2.3 and filling the result buffer in Section 2.9.

4.2. Comparing Domain Decomposition Mergers

The merger is used in the domain decomposition to merge the wavelet structure from different threads into a single. In this work, a merger was explained and implemented. Another merger is implemented in PWM. In Table 4.3, the two different mergers are compared. It can be observed, that except for 20GB, the merger of PWM is always the fastest. Therefore, in the rest of the work, the merger from pwm is used.

4.3. Comparing Shun and Domain Decomposition

In this work two different ways of creating the matrix have been shown. First the way via domain decomposition, and secondly the algorithm following the idea of Shun. In figure 4.4, it can be observed that the domain decomposition is always faster. Additionally, $\tau = 2$ is faster than $\tau = 4$. In the following section, the different internal parts of the algorithm are

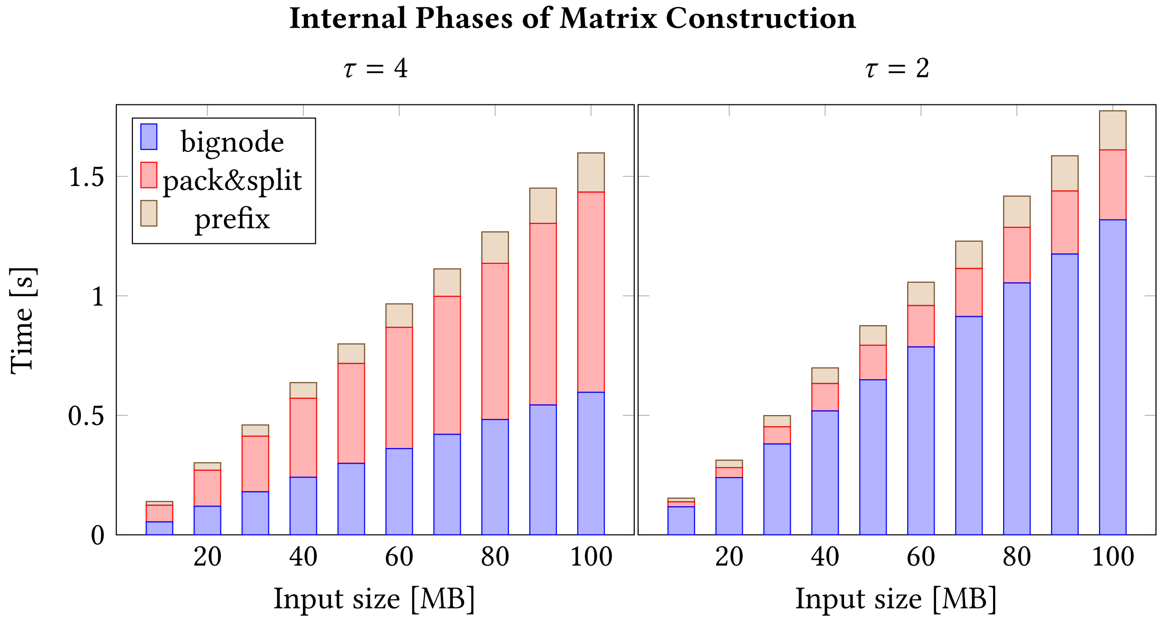


Figure 4.3.: The two plots show the internals of wavelet matrix construction, for $\log\sigma = 8$. On the left side for $\tau = 4$ on the right side $\tau = 2$. The prefix phase is the one creating the histogram, the bignode phase is building the bignode tree. And the pack&split phase transforms the bignode tree into the result object. Finally, this shows that the bignode building takes longer when there is a smaller τ. As input, randomized content with $\log\sigma = 8$ is used.

	Tree		Matrix	
Input size	Thesis Merger	PWM Merger	Thesis Merger	PWM Merger
5*GB*	4207	**4174**	4160	**4014**
10*GB*	8463	**8260**	8353	**8132**
15*GB*	13372	**13300**	12837	**12405**
20*GB*	**17619**	17645	**17328**	17659
25*GB*	22799	**22354**	22663	**21979**
30*GB*	28880	**28025**	27416	**26732**

Table 4.3.: Runtime construction methods, parallelized using domain decomposition, for wavelet structures. The first two columns are for matrix construction, the left construction times for the merger described here, on the right those of the PWM repository. The second two columns are for tree construction. The right one for the matrix construction. As input, randomized content with $\log\sigma = 8$ is used. The columns called *Thesis Merger* is referring to the merger explained in 3.1. The columns called *PWM Merger* are referring to the merger of Dinklage et al. [2]. All values in *ms*.

explained in order to show where the time is spent. Additionally, the difference between τ values are explained.

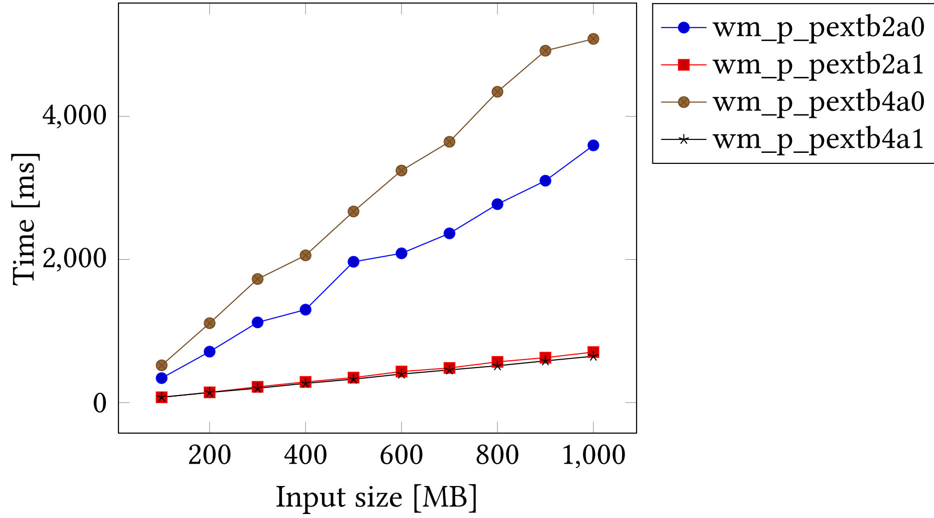


Figure 4.4.: This plot shows the two different parallel matrix construction methods. The *a0* construction algorithm is the one from Shun, and *a1* is the domain decomposition. Running with 64 threads. As input, randomized content with $\log \sigma = 8$ is used.

Comparing Internals with $\tau = 4$. There are 2 tasks dominating the execution time. Construction of the wavelet structures, and the merging. In Figure 4.5 the phases are shown, both for $\tau = 4$. The same τ is selected in order to see the difference between the same amount of *split&sort* calls. The measurements of the phases is the sum of all times, that each thread group takes to perform either merging or splitting. The time is taken between the point where the algorithm goes into parallel execution, until it returns to sequential execution. In the Figure, it can be observed, that merging and splitting scales way worse in the Shun algorithm compared to the Domain decomposition. In order to explain the increased runtime, the overhead added by the Shun approach would need to scale with n. The implementation of Shun makes use of parallelization first for splitting of every single line, then for merging of every single line. This means, that the overhead for synchronizing the end of a thread group is added 8 times instead of only once. This however does not scale with n. Looking at the caching behavior of Shuns approach: When a single line is split, the cache must be first filled with the content of the bignode tree line. After the splitting is done, merging will read all lines from chunks, which will replace the already cached lines of the previous bignode line. The splitting operation of the next line will now need to read again the entire bignode tree line. Comparing this caching behavior with the domain decomposition: When doing splitting of lines in domain decomposition, again each word from the bignode tree line must be read. In the process of splitting, the next bignode tree line is already filled. Now, that the phase of merging is not happening here, the cache will not be rewritten with different words. And the words, that are already in the cache will be reused in the next line to perform the split operations. Therefore the cache utilization is better in the domain decomposition. This overhead also scales with n

and would explain the seen behavior.

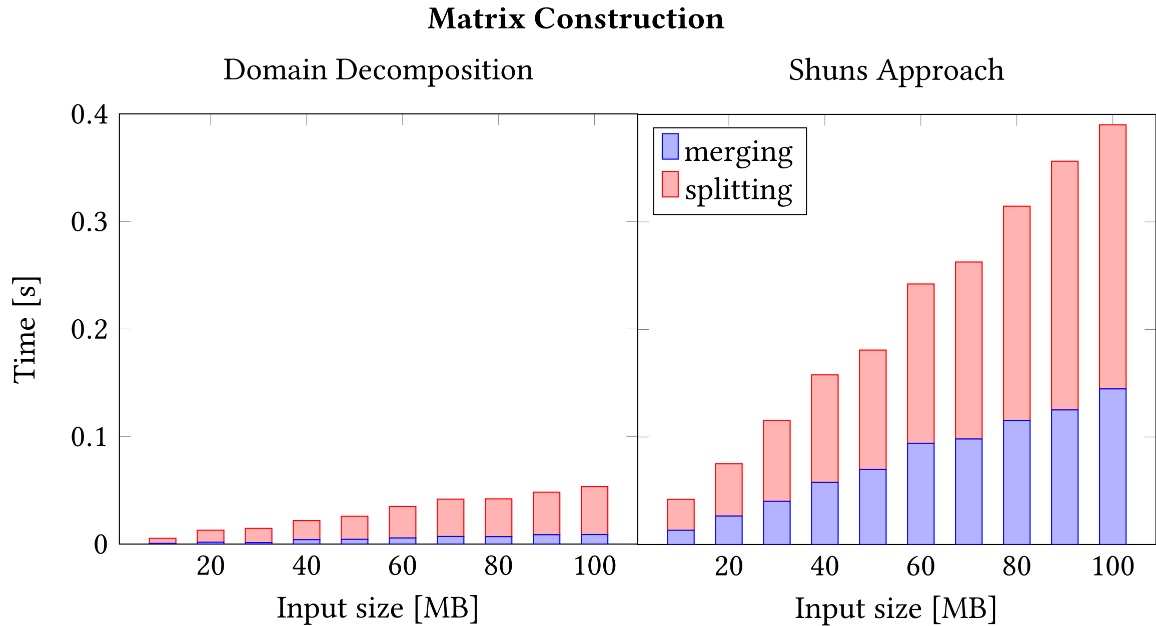


Figure 4.5.: Time spent for *splitting*, and *merging*. In domain decomposition compared to Shuns approach. The construction runs with $\tau = 4$ and 64 threads.

Difference in τ. A major conceptual difference is: In Shuns approach, the entire bignode tree for the entire n is build over the process of constructing the wavelet structure. This is however not necessary, as the bignode tree structure is only required for the result. But is not part of the result. Therefore, the time spent in order to merge the bignode tree into a single object is not an advantage. The value of τ also has an influence on this. A single bignode tree line is bigger with $\tau = 4$ than with $\tau = 2$. Therefore the time spent in merging bignode tree lines is bigger when having $\tau = 4$ compared to $\tau = 2$. Which also explains that $\tau = 4$ is slower than $\tau = 2$

4.4. Comparing to PWM

The fasted available wavelet construction algorithms are currently hosted in the PWM repository. This repository was introduced in [2]. The names of the fastest algorithms are: *pc*, *pc_ss*, and *ps*. Additionally, the parallel versions derived from these are used for comparison.

4.4.0.1. Algorithms from PWM

In this section, the three fastest algorithms from PWM are briefly explained, then, the naive approach *wt_naive* is explained. More details can be found in the code hosted on github. The algorithm *pc* first computes the histogram. Then the all levels are iterated

bottom up. In each level, all the characters are scanned, and the bit at position level is written into the correct position of the result. This can be realized using the position from the histogram. The version called *ps* is doing this similar. However, instead of writing the bit directly into the result buffer, the entire character is stored in a array. Later on, this array is scanned, and the bits are extracted. The algorithm *pc_ss* is walking the entire input, for each character then all the levels are iterated, and the according bit is written into the result buffer. The algorithm *wt_naive* is the slowest and easiest approach. In this approach, each level is iterated. In each level i, 2^i buckets are allocated. Then, all the characters of the input sequence are scanned, the bit of level i is then added to the according result line, additionally, the character is added to the corresponding bucket. The bucket can be calculated by calculating the index of the cell, as it was explained in Section 1.2. Lastly, the different buckets are joined into a single character sequence. This sequence is then replacing the input sequence, and the next levels are continued with this new input.

4.4.1. Comparing using Real World Data

In this section the 5 real word examples are constructed with the construction algorithms *pc*, *pc_ss*, *ps* as well as those from this thesis. The data payloads reflecting real world

Name	$n/10^8$	σ	$\lceil\log\sigma\rceil$
XML	2.9	97	7
DNA	4	16	4
ENG	22.1	239	8
PROT	11.8	27	5
SRC	2.1	230	8
CC	80	243	8
1000G	80	4	2

Table 4.4.: These payloads are transformed into an alphabet without unused characters. This results in a more compact alphabet, where the $\log\sigma$ value has a direct impact on the depth of the levels that need to be walked. The script for transforming the input can be found in the repository of this work. From payload *CC* and *1000G*, only a 80GB prefix is used, as the original payload was too big for the available hardware.

usages are showed in Table 4.4. Notably, two payloads do have $\log\sigma$ values that are not evenly dividable by τ. For these payloads normal algorithms, that do not using bignode trees have an advantage. Due to the nature of the algorithm explained in this work always τ bits per character have to be handled. Therefore the payload *XML* has the same runtime as running with a $\log\sigma = 8$ and *PROT* $\log\sigma = 6$. The construction is evaluated for $\tau = 4$ and $\tau = 2$.

Sequential Algorithms Comparison for Real Data. For comparing PWM and this work, tree and matrix construction times are evaluated and compared in Figure 4.6 and Figure 4.6. One of the bigger differences from PWM to the this work is that $0 \not\equiv \lceil\log\sigma\rceil \mod \tau$ results in an overhead. Therefore the advantage of the here explained algorithm is lower for *prot*.

For matrix construction this disadvantage is bigger, so the PWM algorithms are faster. As the size of *1000G* and *cc* are quite big, they are skipped for sequential construction.

algo	*xml*	*dna*	*eng*	*prot*	*src*
wt_naive	11750	9614	112095	32322	9252
wt_pc	5240	4377	49044	14360	4461
wt_pc_ss	5311	3933	46109	**13953**	4367
wt_pextb2a1	5130	**3592**	36362	14684	3488
wt_pextb4a1	**4648**	3900	**33924**	15875	**3328**
wt_ps	5573	5119	54687	15716	4849

Table 4.5.: Comparison of constructing wavelet tree times for real datasets. The *pext* algorithm is the cell iterator approach. Each result in *ms*.

algo	*xml*	*dna*	*eng*	*prot*	*src*
wm_naive	11916	10790	113568	41676	10364
wm_pc	5217	4369	49148	14441	4447
wm_pc_ss	5298	3825	45977	**13782**	4329
wm_pextb2a0	5227	**3798**	37050	15816	3526
wm_pextb4a0	**4537**	3949	**32835**	16413	**3233**
wm_ps	5534	5117	54713	15518	4825

Table 4.6.: Comparison of constructing wavelet matrix times for real datasets. Each result in *ms*.

Parallel Algorithms Comparison for Real Data. The results of the parallel constructions, in Table 4.4.1, do look similar as ones from sequential ones. It can be observed, that wavelet tree construction for *DNA*, *ENG*, *SRC*, *CC*, and *1000G* payloads, the *pext* versions are the fastest. For wavelet tree construction with *pext*, the payload *XML* is also faster compared to PWM's algorithms. For wavelet matrix constructions, PWM's *wm_dd_ps* algorithm is faster. In this case, the disadvantage with having to iterate one more level compared to PWM cases this. However, the speed difference is only *0,4%*. The payload *PROT* is being constructed faster with *PWM* algorithms. For this payload, the disadvantage with having to iterate one more level is also applying. In the next section randomly generated payloads are used to measure construction times.

algo	*xml*	*dna*	*eng*	*prot*	*src*	*cc*	1000*G*
wt_dd_pc	221	172	1980	684	166	78973	**13879**
wt_dd_pc_ss	228	176	1795	666	169	75872	19930
wt_dd_ps	207	172	1906	**457**	160	72142	25380
wt_p_pextb2a3	202	166	1806	618	**147**	69646	23917
wt_p_pextb4a3	**188**	**151**	**1765**	894	154	**67833**	28815
wt_ppc	1100	1495	8153	4038	1120	271403	258318
wt_ppc_ss	1071	2141	7982	4254	866	292911	247550
wt_pps	462	470	3409	1439	362	128893	74403

Table 4.7.: Comparing construction times for parallel tree creation. Each result in *ms*.

algo	*xml*	*dna*	*eng*	*prot*	*src*	*cc*	1000*G*
wm_dd_pc	218	169	1750	**475**	156	82242	23481
wm_dd_pc_ss	226	173	1786	868	168	76889	**17824**
wm_dd_ps	**211**	170	1889	501	154	69996	26321
wm_p_pextb2a2	212	**158**	1858	631	152	72222	24485
wm_p_pextb4a2	213	173	**1738**	815	**139**	**68031**	29451
wm_ppc	1117	1509	8135	4038	760	288285	256209
wm_ppc_ss	1116	1459	8067	4195	801	292421	248635
wm_pps	442	468	3401	1458	339	127339	76339

Table 4.8.: Comparing construction times for parallel matrix creation.Each result in *ms*.

Strong scaling and weak scaling In Figure 4.6, weak and strong scaling experiments for the payloads *cc*, *1000G*, and *eng* are plotted. It can be observed, that *1000G* has a lot higher throughput compared to the other two payloads. Roughly $1.25 * 10^6 \frac{\text{Byte}}{\text{ms}}$ with *eng* and *cc*, and roughly $3 * 10^6 \frac{\text{Byte}}{\text{ms}}$ for *1000G*. The difference between those is, that $\log \sigma$ of 1000G is 2, while those of *eng* and *cc* is 8. Comparing the throughput to the algorithms of PWM shows a similar situation as Figure 4.8 and Figure 4.4.1. The payloads *eng*, and *cc* do have a higher throughput for all thread configurations, except 30 threads. For the payload *1000G*, PWM constructions are faster.

wm_dd_pc
wm_dd_pc_ss
wm_dd_ps
wm_p_pextb2a2
wm_p_pextb4a2
wt_dd_pc
wt_dd_pc_ss
wt_dd_ps
wt_p_pextb2a3
wt_p_pextb4a3

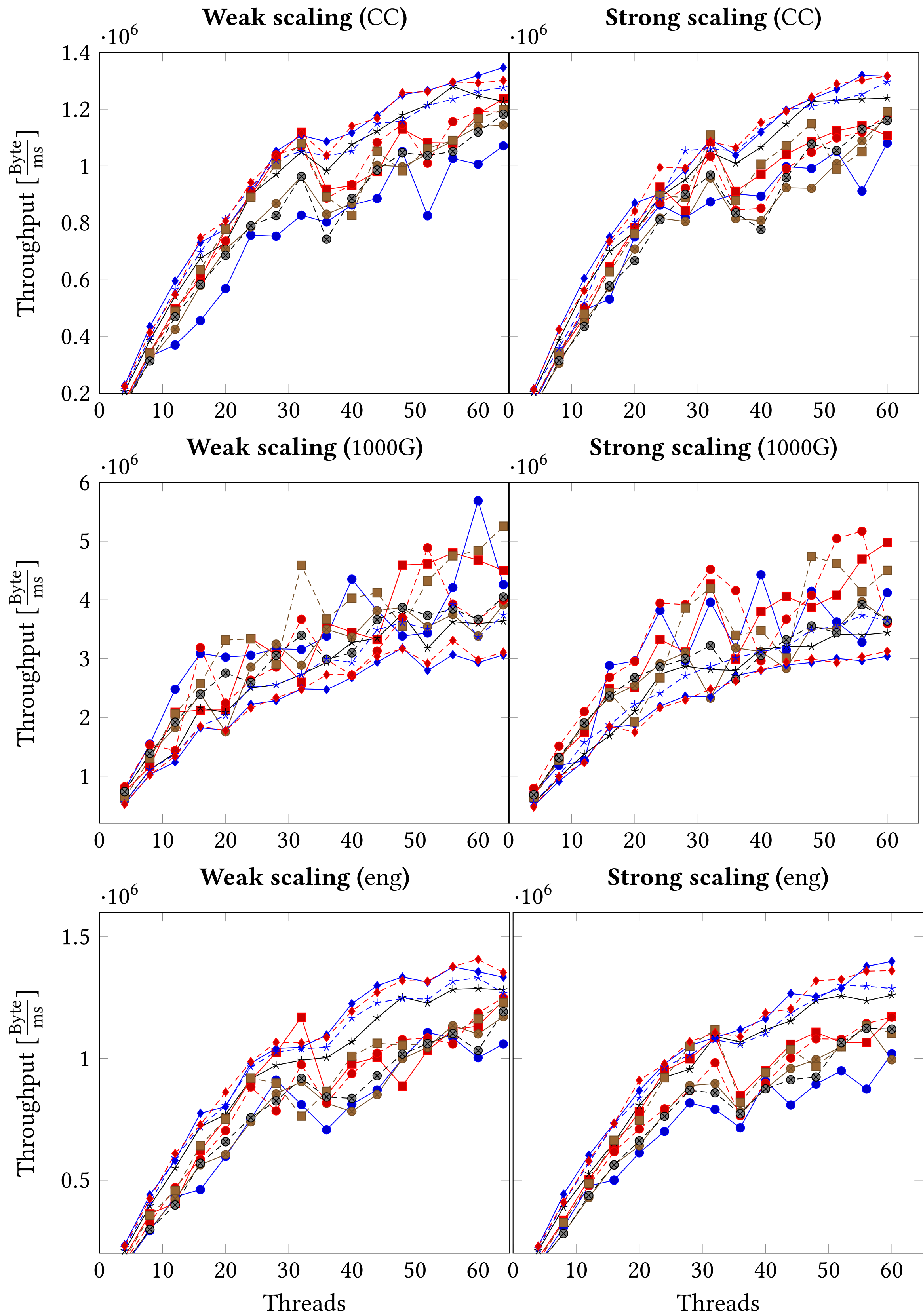


Figure 4.6.: Weak and strong scaling experiments for the real world data. In weak scaling, 100M per thread are added. In strong scaling 6400MB are performed.

4.4.2. Comparing using Generated Data

In this section construction times for randomly generated payloads are compared. The payloads do have sizes from *1GB* to *3GB* with a step size of *5GB*.
For better benchmarks, each generated payload is copied and processed that each sized payload $\log\sigma \in \{2, 3, 4, 5, 6, 7, 8\}$. The benchmarks are run with *pc*, *pc_ss*, and *ps* and its parallel versions. As well as *pext* algorithms and the median runtime over 25 iterations is measured.

Sequential Comparison. In Tables 4.9 and 4.10 the comparison of sequential construction times can be seen. Each cell of the table contains the speed up, which is calculated by dividing the fastest PWM runtime with the fastest *pext* runtime. In each column titled $speedup_x$, x is expressing the $\log\sigma$ from the available payloads. What can be observed in general, is that the tree construction is having a higher speedup than the matrix construction. The speedup decreases with a decreasing $\log\sigma$. Additionally, uneven $\log\sigma$ are slower than the even predecessor. Finally, for tree construction, $\log\sigma < 4$ results in a speedup below 1 which means that PWM is again faster. In matrix construction this is already happening at $\log\sigma = 5$. The general decrease in speedup, for a decreasing $\log\sigma$ can be explained by the fact that the *pext* algorithms are using two phases. The algorithm runtime is heavily depending on the choice of τ, which can be used to perform more work in the first phase, or in the second phase. This has already been shown in Section 4.2, it also showed the choice of τ is depending on σ. For lower σ values, τ was also set to a lower value. However, τ cannot be set lower than 2, if now the $\log\sigma$ value gets too low, not enough speedup can be realized by parallel bit extraction in the second phase. Therefore, an algorithm not having a σ depended property can scale much better, which is the case for PWM. Hence PWM scales better for a $\log\sigma < 4$.
The higher performance degredation for uneven $\log\sigma$s can be explained again, by the choice of τ, and the resulting implications. As shown in 4.2, $\tau = 4$ is used for $\log\sigma = 8$ and $\tau = 2$ for everything else. Just as a reminder: τ cannot be smaller than 2 as that would not allow parallel extraction using *pext*. Also, when $\tau = 4$ is selected, always 4 bits at once are copied when the bignode tree is build. This means, that for a $\log\sigma \mod \tau \neq 0$, at least one bit too much is going to be copied into the bignode tree. This also results in a less efficient *pack* phase. In PWM algorithms, the iteration can simply be stopped at a higher depth, which results in fewer memory writes.

Parallel Comparison. The parallel construction algorithms in Table 4.11 show similar results as in the real world data comparison. For $\lceil\log\sigma\rceil > 7$ the construction is faster then PWM. And only $\lceil\log\sigma\rceil = 4$ is again a little bit faster than PWM. However, for increasing size, the speedup is decreasing while staying bigger than 1.

Strong scaling and weak scaling. Two scaling experiments often done with parallel algorithms are so called strong scaling and weak scaling experiments.
In a strong scaling experiment, the number of threads is varying while the problem size stays constant. For a weak scaling experiment, the number of threads is varying as well, however, the problem size per thread stays the same. This means, that $\frac{\text{problem_size}}{\text{threads}}$ stays

Payload	s_8	s_7	s_6	s_5	s_4	s_3	s_2
100MB	1.3120	1.1274	1.2488	1.0122	1.1162	0.7991	0.7374
200MB	1.3473	1.1424	1.2407	1.0297	1.1219	0.8027	0.7360
300MB	1.3082	1.1221	1.2426	1.0180	1.1272	0.8042	0.7276
400MB	1.3546	1.1598	1.2564	1.0313	1.1253	0.8079	0.7298
500MB	1.3348	1.0857	1.2426	1.0117	1.1162	0.8006	0.7259
600MB	1.3077	1.1451	1.2450	1.0264	1.1193	0.7990	0.7218
700MB	1.2963	1.1296	1.2522	1.0353	1.1179	0.7996	0.7238
800MB	1.3509	1.1423	1.2635	1.0349	1.1277	0.7965	0.7232
900MB	1.3124	1.1186	1.2457	1.0253	1.1202	0.8041	0.7247
1000MB	1.3478	1.1513	1.2542	1.0217	1.1238	0.7995	0.7217

Table 4.9.: Sequential speedups of wavelet tree creation and the fastest PWM algorithm, with payloads with different $\log \sigma$. Each column is described as $s_{\lceil \log \sigma \rceil}$. With 64 threads.

Payload	s_8	s_7	s_6	s_5	s_4	s_3	s_2
100MB	1.2958	1.1121	1.1522	0.9429	1.0500	0.7525	0.7487
200MB	1.3088	1.1292	1.1496	0.9490	1.0564	0.7518	0.7406
300MB	1.2756	1.1014	1.1465	0.9353	1.0516	0.7515	0.7348
400MB	1.3446	1.1446	1.1573	0.9412	1.0489	0.7516	0.7402
500MB	1.2734	1.1081	1.1380	0.9313	1.0621	0.7434	0.7299
600MB	1.3265	1.1381	1.1523	0.9462	1.0458	0.7463	0.7288
700MB	1.2913	1.0815	1.1082	0.9324	1.0491	0.7434	0.7296
800MB	1.3098	1.1164	1.1444	0.9546	1.0355	0.7453	0.7293
900MB	1.2987	1.1016	1.1405	0.9415	1.0482	0.7450	0.7439
1000MB	1.3246	1.1247	1.1427	0.9444	1.0596	0.7558	0.7577

Table 4.10.: Sequential speedups of wavelet matrix creation and the fastest PWM algorithm, with payloads with different $\log \sigma$. Each column is described as $s_{\lceil \log \sigma \rceil}$. With 64 threads.

constant.
In the Figure 4.7 the weak scaling experiment is plotted. In Figure 4.8 the strong scaling experiment is plotted. It can be observed, that the throughput in the strong scaling has a dent at around 35 threads. This dent also stays over several benchmark runs. Comparing the PWM algorithms to the *pext* constructions do show, that the *pext* versions have a higher throughput, except for 30 threads. This observation is also the same for the real data payloads.

Payload	s_8	s_7	s_6	s_5	s_4	s_3	s_2
$100MB$	1.0132	1.1471	0.8305	0.6290	1.0465	0.6829	0.6154
$200MB$	1.0138	1.1221	0.8750	0.7037	1.0633	0.6933	0.6596
$300MB$	1.1466	1.1058	0.8704	0.7063	1.0000	0.6579	0.5658
$400MB$	1.0861	0.9751	0.8804	0.6729	1.0625	0.5519	0.6129
$500MB$	1.0804	1.0726	0.8702	0.7099	1.0162	0.6720	0.6316
$600MB$	1.1526	1.0806	0.9042	0.6977	1.0000	0.6682	0.6277
$700MB$	1.1556	1.1372	0.9158	0.6720	0.9960	0.6824	0.5754
$800MB$	1.1152	1.0233	0.9252	0.7536	0.9731	0.7517	0.6378
$900MB$	1.1655	1.0718	0.9236	0.7228	1.0636	0.6873	0.6093
$1GB$	1.1371	1.0836	0.9267	0.7446	1.0241	0.8000	0.8227
$5GB$	1.0415	0.9170	0.9585	0.8191	0.8726	0.6008	0.8062
$10GB$	1.0776	0.9133	0.9463	0.7787	0.8698	0.6187	0.5836
$15GB$	1.1195	0.9415	0.9671	0.7890	0.7993	0.6114	0.5624
$20GB$	1.0276	0.8927	1.0254	0.7823	0.8091	0.6245	0.6301
$25GB$	1.0449	0.8653	0.9487	0.7642	0.8820	0.7002	0.5658
$30GB$	1.0560	0.8487	1.0288	0.8526	0.9058	0.5997	0.6130
$35GB$	1.0854	0.9067	0.9570	0.7721	0.8133	0.6982	0.6752
$40GB$	1.0920	0.8873	0.9279	0.7850	0.8417	0.6647	0.7202
$45GB$	1.0689	0.8906	0.9076	0.7359	0.7849	0.6524	0.6706
$50GB$	1.1030	0.8359	0.9802	0.7849	0.7990	0.5624	0.7091

Table 4.11.: Parallel speedups for wavelet tree creation with payloads with different $\log \sigma$. Each column is described as $s_{\lceil \log \sigma \rceil}$. With 64 threads. Further details can be received from A.1.

4.5. Memory Consumption

In order to measure the memory consumption, the allocation calls to *malloc*, *calloc*, and *free* are counted, and the size of the allocated memory is summed up. As already explained, there are 4 different allocations: for the histogram, the bignode tree, the result buffer, and the two swapping lines. The histogram size is only depending on $\log \sigma$. The bignode tree is always exactly the size of the input n. The result buffer is as well as big as n. The two swapping lines are of the size $2\tau n/8$. This means, that for $\tau = 2$, $2.5n$ are required, and for $\tau = 4$ $3n$ are required. In Figure 4.9 the two clusters can be seen at using $3.5n$ and $4n$. The last missing n that is required in order to explain the results, is the actual file that is loaded into memory. The memory allocated for the histogram is not visible in this plot, as it is too small in contrast of the input size.
The domain decomposition also stays within the the cluster of its underlying algorithm. Even though the peak is measured, the additional result buffer allocated is not added to the peaks top, as the bignode tree memory object is destroyed while the underlying algorithm is finished.
Additionally to the two clusters, that explain the dependency on τ, it is also visible that the domain decomposition does not require more memory than the normal sequential version of the algorithm.

Payload	s_8	s_7	s_6	s_5	s_4	s_3	s_2
100MB	0.9747	1.0986	0.8167	0.7049	0.8182	0.6667	0.6000
200MB	1.1203	1.0211	0.7881	0.9138	1.0723	0.6582	0.6739
300MB	1.0140	1.0833	0.8758	0.7669	1.0348	0.6552	0.6522
400MB	1.0878	1.0303	0.9352	0.7411	1.0196	0.6846	0.5728
500MB	1.1191	1.0495	0.8425	0.7189	1.0000	0.6631	0.6637
600MB	1.0682	1.1018	0.8415	0.7378	0.9690	0.7589	0.6312
700MB	1.1069	1.0243	0.8892	0.7383	0.9768	0.6517	0.6398
800MB	1.0749	1.0558	0.8712	0.7773	0.9831	0.6711	0.6571
900MB	1.0753	1.0952	0.8703	0.7562	0.9231	0.6395	0.8182
1GB	1.1222	1.0382	0.9046	0.7650	1.0186	0.6603	0.6318
5GB	1.0785	0.9428	0.8972	0.7430	0.9748	0.5738	0.6562
10GB	1.0692	0.9474	0.9451	0.7712	0.7807	0.6725	0.7201
15GB	1.0766	0.8371	0.9671	0.8084	0.7584	0.6499	0.5933
20GB	1.0028	0.9344	1.0031	0.7902	0.8611	0.6404	0.7279
25GB	1.0483	0.8581	0.9270	0.7967	0.9884	0.6600	0.6894
30GB	1.0372	0.8663	0.9081	0.7553	0.8417	0.6147	0.7804
35GB	1.1120	0.9488	0.9045	0.8408	0.9421	0.7050	0.7523
40GB	1.0315	0.9148	0.8963	0.8284	0.8910	0.6640	0.6674
45GB	1.1092	0.8235	0.9874	0.8549	0.8818	0.6339	0.7002
50GB	1.0806	0.9007	0.9081	0.8031	0.8568	0.7077	0.8657

Table 4.12.: Parallel speedups for wavelet matrix creation with payloads with different $\log \sigma$. Each column is described as $s_{\lceil \log \sigma \rceil}$. With 64 threads. Further details can be received from A.1.

Shun Parallelization. The memory usage of the Shun parallelization requires more memory than the normal domain decomposition. While building a matrix, each line of chunks requires $p(3\frac{n\tau}{p}) = 3n\tau$ bits and $\frac{3n\tau}{8}$ bytes of memory. These are required to the additional memory allocations from the previous paragraph. Therefore there are for $\tau = 2$ an expected memory usage of $4.25n$ and for $\tau = 4$ a memory usage of $5.5n$

Comparison to PWM. There are 6 clusters in the plot, 4 are from the algorithms explained in this work, the other 2 are from PWM, there are two cluster, one is residing at $2n$ the other is located at $3n$. The cluster at $2n$ is consisting of: *ppc_ss*, *pc_ss*, *ppc*, *pc* and the $3n$ contains: *dd_pc_ss*, *dd_pc*, *dd_ps*, *pps*, *ps*.
The domain decomposition here does move the two construction methods *pc_ss* and *pc* from $2n$ to $3n$. This is here happening, because after the construction inside the domain decomposition, no freeing of any memory object big enough happens. This effect is happening for the *ps* algorithm, however, this one is already in $3n$.

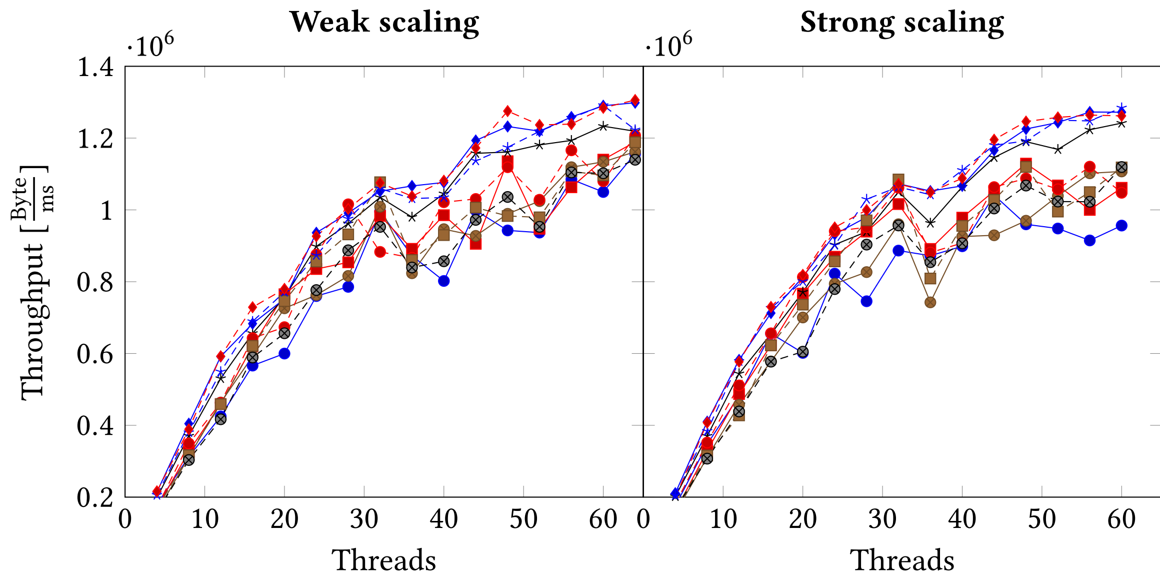
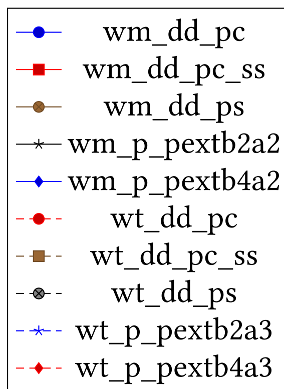


Figure 4.7.: Weak scaling experiment, per thread 100MB are assigned. In each step 4 more threads are added.

Figure 4.8.: Strong scaling experiment with 6,4GB of payload. With each step 4 more threads are added.



4.6. Intrinsic Functions

In this work the two intrinsic functions are used. *pext*, and *popcnt*. *popcnt* is used to answer query calls, and count bits in the mask $fill_x$, which is used in *split&sort*. *pext* instructions are used to extract bits in the *split&sort* phase, as well in the *pack* phase. When the implementation of this work started, initially every benchmark was ran on *AMD Ryzen 7 3800X 8-Core Processor*. It turned out that the algorithms using *pext* were a lot slower than PWM ones. Researching the exact instructions for AMD brought up [1] that these instructions are implemented as microcode in certain chips, which results in a 18 cycle latency.

[1] AMD instruction wikipedia `https://en.wikipedia.org/wiki/X86_Bit_manipulation_instruction_set`.

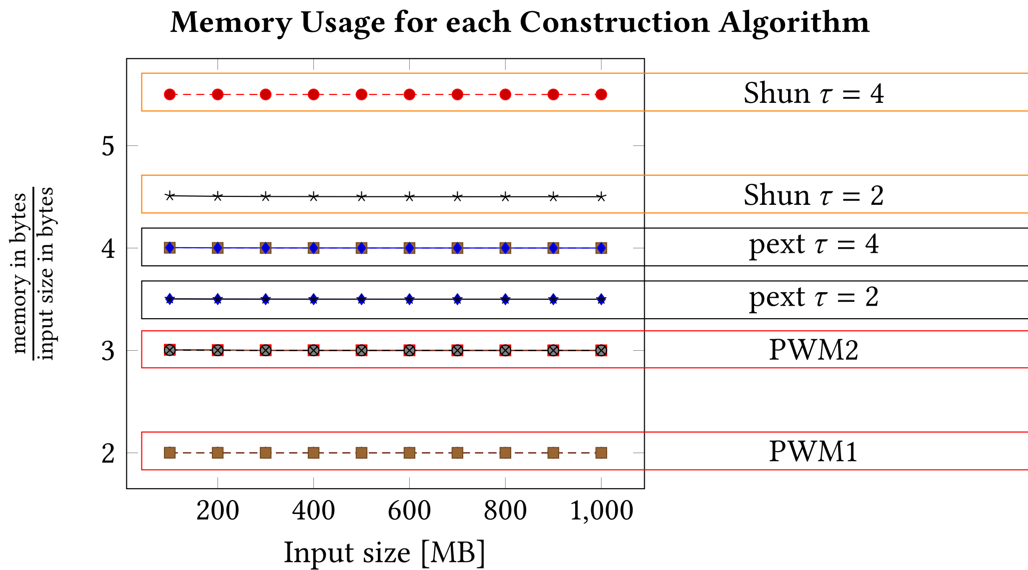


Figure 4.9.: Plot of memory usage divided by the payload size. The plot shows 6 cluster. Two clusters are from the PWM algorithm. Two from the normal *pext* algorithm and their domain decomposition versions. The last 2 clusters are from the Shun parallelization. For parallel constructions, 64 threads are used.

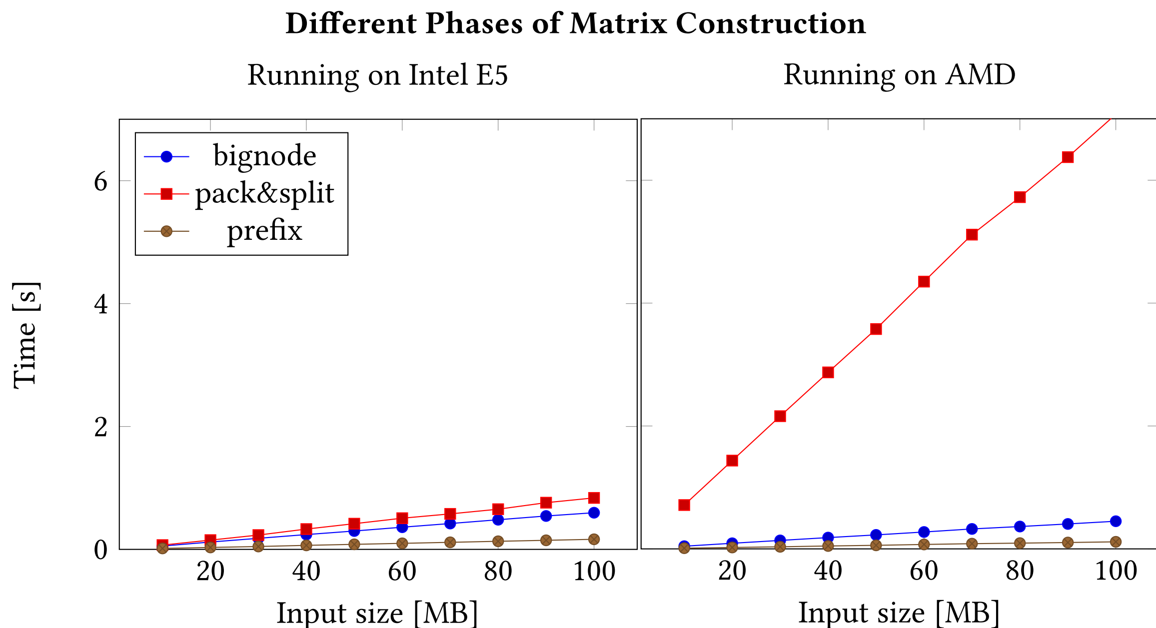


Figure 4.10.: Plots showing the difference between the internal phases of the construction on a Intel E5 CPU on the left, and of the AMD CPU on the right.

These speed differences are also visible in the plot at Figure 4.10, which displays the timing of sequential construction of matrix. The phase *pack&split*, which is the only one utilizing

pack&split, can be seen to be roughly 10 times higher on the AMD side, compared to the Intel side.

4.7. Code-Size

This work has now compared memory and runtime. Another interesting property is the code size. The code sizes of the different algorithms from PWM and this work are compared, for comparing them, the LOC of all called methods are compared. Methods from c++ like vector construction is not counted. In Table 4.13 it can be seen, that histogram building is roughly the same between this work, and PWM. The bignode phase of *pext* algorithms is comparable to the flushing of the PWM repositories, conceptional as well as judging by the size. However, flushing the bignode tree into the final results requires another 93 lines of code. This shows that using any form of parallel processing of bits, either by using *pext* or by using tables as described in [4] or [10], results in an significant increase in complexity. The code size of the parallel versions is only having an impact

Algorithm	Histogram [LOC]	Bignode [LOC]	Result filling [LOC]
pc	23	-	41
pc_ss	23	-	28
ps	23	-	28
pext	26	25	93
shun	26	169	269

Table 4.13.: Different code sizes compared in lines. For *pext*, cell iteration for tree building is evaluated. The phase *flushing* is constructing the result buffer. For PWM algorithm this is the normal insertion of bits according to *pc*, *ps*, or *pc_ss*. For *pext*, *Shuns* algorithms this is the transformation from bignode tree to result.

on the none domain decomposition versions, as the merging of the wavelet structure is performed by the same code, and has no difference in the conceptional working. The Shun [8] approach is again a lot bigger in its code size, as a lot more different merging is performed.

4.8. Runtime prediction for dynamic τ setting

In chapter 2.3, an alternative approach to Kaneta [9], Munro et al. [4], and Babenko et al. [10] was explained. This algorithm could not be implemented due to the lack of time, and the point of time when this approach started to form.
The other algorithms have been evaluated using a range of different τs. Therefore, the time spend per line using a specific τ can be calculated. In the experiments, a payload with $\lceil \log \sigma \rceil = 8$ has been used. Therefore the time spend in a single line can be calculated by dividing the time spend for the entire construction by 8. In the explanation of the algorithm, it is explained that for line $4 \leq i < 8$ $\tau = 8$ is used. For $2 \leq i < 4$ $\tau = 4$ is used.

And finally for $0 \leq i < 2$ $\tau = 2$ is used. This means, that the time can be calculated using:

$$T_{dyn} = \frac{1}{2} \cdot t_8 + \frac{1}{4} \cdot t_4 + \frac{1}{4} \cdot t_2 \tag{4.1}$$

With t_x taken from the run with $\tau = x$. Using T_{dyn} as a prognoses, the Table 4.8 shows what runtime is to be expected. This prognoses shows, that this approach would roughly be faster than the current implementations. And it replaces the entire bignode creation, which lowers the amount of code required for the algorithm. Additionally, no τ must be selected for a given input configuration, which makes the usage of the algorithm easier.

size	$\tau = 2$	$\tau = 4$	$\tau = 8$	T_{dyn}	*pext*
10MB	0.0207	0.0698	0.1748	0.1255	0.1396
20MB	0.0418	0.1508	0.3700	0.2643	0.3041
30MB	0.0718	0.2329	0.5675	0.4066	0.4633
40MB	0.1147	0.3306	0.8073	0.5804	0.6432
50MB	0.1447	0.4180	0.9754	0.7099	0.8064
60MB	0.1730	0.5074	1.2309	0.8835	0.9753
70MB	0.2012	0.5775	1.3775	0.9978	1.1228
80MB	0.2327	0.6537	1.5257	1.1151	1.2790
90MB	0.2639	0.7599	1.8271	1.3168	1.4631
100MB	0.2926	0.8376	1.9822	1.4372	1.6117

Table 4.14.: Table showing the prognoses for the alternative algorithm approach described. The prognoses are calculated using Formula 4.1. The input values t_8, t_4, and t_2 are used from measuring the internals of the algorithms with $\tau = 8$, $\tau = 4$, and $\tau = 2$. All values in seconds

5. Conclusion

Wavelet structures can be used to answer select, access, and rank queries. These queries can be used to process and work with strings. Wavelet structures can be of two kinds: Trees and matrices. The construction of both takes a similar time, and the implementation of select, access, and rank queries is similar. In the work of Dinklage et al. [2], construction algorithms have been published in a github repository. These algorithms build these structures in $O(n \log \sigma)$. Munro et al. [4] and Babenko et al. [10] showed, that the runtime can be minimized to $O\left(n \left\lceil \log \sigma / \sqrt{\log n} \right\rceil\right)$ by utilizing tables. The utilization of tables results in a speedup utilizing bit word parallelization. Kaneta implemented this approach in [9] replacing the table lookup with the intrinsic instructions like *pext*, *pshufl*. These instructions are explained in Section 1.3. In this work, we implemented these construction algorithms again using the *pext* instruction. We then used these implementations to evaluate the performance for parallelization using domain decomposition. Additionally to that, the idea from Shun [8] was implemented as well.
In this work, a second implementation of the wavelet structure merger from PWM was produced. Lastly it showed that the PWM merger is mostly the fastest.

5.1. Conclusion over *pext* algorithms

With the idea of Munro et al. and Babenko et al. two additional parameter was added. In Munro et al. and Babenko et al. one parameter, called β, is defining how long the keys of the key value pairs of the tables are. In these works, this parameter could simply be chosen to minimize the theoretical runtime. However, in case of using *pext*, this is defined by the hardware instruction, and the length of the registers. The second parameter is τ, is controlling how many bits can be processed in parallel within a single word. It turned out that this parameter has quite a big influence on the runtime, and must be chosen with a dependency on $\log \sigma$.

Sequential Version. The sequential *pext* construction algorithms heavily depend on $\log \sigma$. For $\log \sigma = 8$, it turned out to be faster than the PWM algorithms, this is detailed in Figures 4.5 and 4.6. For a lower $\log \sigma$ the speedup constantly decreases, lastly, for lower $\log \sigma$ values, the PWM construction algorithms are again faster. Over the process of implementing all this, it turned out, that there are different implementation from the *pext* instruction on a hardware level, this is detailed in 4.6

Parallel Version. The parallel versions of the algorithms turned out to also outperform the PWM algorithms for $\log \sigma = 8$, however, by a lot lower margin. The speedup also decreases way faster, and the PWM algorithms are faster at a higher $\log \sigma$ compared to the sequential versions.

The implementation of the Shun approach showed, that it cannot be faster than the domain decomposition versions. A more detailed analysis of that can be found in Section 4.5.

5.2. Further Improvements

In the algorithms implemented here, only 64 bit registers have been used for *pext* instructions. However, it would be a good idea to elaborate if the *pext* instructions can be used with 128 bit registers, as this would increase the amount of parallel extraction that can be used to implement the two main function *split&sort* and *pack*.
Additionally, in Section 2.3 a new approach for implementing the construction of wavelet structures has been presented. This elaborates the fact, that we can choose a free τ between different lines. This is only possible due to using the *pext* instruction. In the approach from Munro et al. and Babenko et al., it would require all tables to be recreated.
Further more, certain τ and $\log \sigma$ configurations would allow for further simplifications of the bignode tree creation. For example: $\tau = 2$ and $\log \sigma = 2$ would mean, that the bignode tree phase only consists of copying two bits from each character into the bignode tree. This could also be implemented using *pext*, utilizing again bit word parallelization.
The work of Dinklage et al. [2] showed different approaches for walking the input. These different approaches resulted in different run times. This knowledge could be used to optimize bignode building further more.

Bibliography

[1] Brent, Richard P. "The Parallel Evaluation of General Arithmetic Expressions". In: *J. ACM* (1974). DOI: 10.1145/321812.321815.

[2] Dinklage, Patrick and Ellert, Jonas and Fischer, Johannes and Kurpicz, Florian and Löbel, Marvin. "Practical Wavelet Tree Construction". In: *ACM J. Exp. Algorithmics* (2021). DOI: 10.1145/3457197.

[3] George, Thomas and Sarin, Vivek". "Domain Decomposition". In: *Encyclopedia of Parallel Computing*. Ed. by David Padua. Boston, MA: Springer US, 2011, pp. 578–587. ISBN: 978-0-387-09766-4. DOI: 10.1007/978-0-387-09766-4_291.

[4] J. Ian Munro and Yakov Nekrich and Jeffrey Scott Vitter. "Fast construction of wavelet trees". In: *Theor. Comput. Sci.* 638 (2016), pp. 91–97. DOI: 10.1016/j.tcs.2015.11.011.

[5] José Fuentes-Sepúlveda and Erick Elejalde and Leo Ferres and Diego Seco. "Parallel construction of wavelet trees on multicore architectures". In: *Knowl. Inf. Syst.* 51.3 (2017), pp. 1043–1066. DOI: 10.1007/s10115-016-1000-6.

[6] Joseph F. JáJá. *An Introduction to Parallel Algorithms*. Addison-Wesley, 1992. ISBN: 0-201-54856-9.

[7] Julian Labeit and Julian Shun and Guy E. Blelloch. "Parallel lightweight wavelet tree, suffix array and FM-index construction". In: *J. Discrete Algorithms* 43 (2017), pp. 2–17. DOI: 10.1016/j.jda.2017.04.001.

[8] Julian Shun. "Improved parallel construction of wavelet trees and rank/select structures". In: *Inf. Comput.* 273 (2020), p. 104516. DOI: 10.1016/j.ic.2020.104516.

[9] Kaneta, Yusaku. "Fast Wavelet Tree Construction in Practice: 25th International Symposium, SPIRE 2018, Lima, Peru, October 9-11, 2018, Proceedings". In: Jan. 2018, pp. 218–232. ISBN: 978-3-030-00478-1. DOI: 10.1007/978-3-030-00479-8_18.

[10] Maxim A. Babenko and Pawel Gawrychowski and Tomasz Kociumaka and Tatiana Starikovskaya. "Wavelet Trees Meet Suffix Trees". In: *SODA*. SIAM, 2015, pp. 572–591. DOI: 10.1137/1.9781611973730.39.

[11] Peter Sanders and Kurt Mehlhorn and Martin Dietzfelbinger and Roman Dementiev. *Sequential and Parallel Algorithms and Data Structures - The Basic Toolbox*. Springer, 2019.

[12] Pramod Chandra P. Bhatt and Krzysztof Diks and Torben Hagerup and V. C. Prasad and Tomasz Radzik and Sanjeev Saxena. "Improved Deterministic Parallel Integer Sorting". In: *Inf. Comput.* 94.1 (1991), pp. 29–47. DOI: 10.1016/0890-5401(91)90031-V.

[13] *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms, January 12-14, 2003, Baltimore, Maryland, USA*. ACM/SIAM, 2003. ISBN: 0-89871-538-5. URL: http://dl.acm.org/citation.cfm?id=644108.

[14] Vishkin, Uzi. *Thinking in Parallel: Some Basic Data-Parallel Algorithms and Techniques.* URL: https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.8.3584&rep=rep1&type=pdf.

A. Appendix

A.1. Parallel wavelet construction using generated data

Algorithm	8	7	6	5	4	3	2
$wm_dd_pc_{50GB}$	51601.0	42858.0	37529.0	28994.0	21428.0	19715.0	13007.0
$wm_dd_pc_ss_{50GB}$	49156.0	39991.0	30800.0	26776.0	21086.0	16855.0	13519.0
$wm_dd_ps_{50GB}$	47647.0	39712.0	34247.0	27728.0	23955.0	19105.0	17769.0
$wm_p_pextb2a2_{50GB}$	44092.0	44091.0	33918.0	33341.0	24611.0	23815.0	15025.0
$wm_p_pextb4a2_{50GB}$	46347.0	46275.0	38856.0	37665.0	27232.0	25030.0	18809.0
wm_ppc_{50GB}	162224.0	160064.0	161627.0	157670.0	172909.0	196855.0	155791.0
$wm_ppc_ss_{50GB}$	200992.0	210481.0	200871.0	217146.0	180143.0	184415.0	152028.0
wm_pps_{50GB}	78795.0	78593.0	74302.0	62202.0	61529.0	52019.0	47234.0
$wt_dd_pc_{50GB}$	48240.0	38747.0	34016.0	29364.0	19323.0	13602.0	11790.0
$wt_dd_pc_ss_{50GB}$	48634.0	35947.0	31965.0	25002.0	20279.0	18442.0	11313.0
$wt_dd_ps_{50GB}$	48058.0	39775.0	34657.0	29087.0	22291.0	17406.0	16005.0
$wt_p_pextb2a3_{50GB}$	43572.0	43005.0	32610.0	31853.0	24183.0	24186.0	15955.0
$wt_p_pextb4a3_{50GB}$	45303.0	44619.0	36923.0	36603.0	26144.0	25356.0	18055.0
wt_ppc_{50GB}	177602.0	166174.0	155719.0	157171.0	203019.0	198977.0	156542.0
$wt_ppc_ss_{50GB}$	203858.0	197630.0	205344.0	193890.0	178974.0	181599.0	174992.0
wt_pps_{50GB}	78181.0	78822.0	72600.0	64694.0	59171.0	52045.0	48685.0
$wm_dd_pc_{40GB}$	33287.0	32361.0	27022.0	23098.0	17134.0	12584.0	10400.0
$wm_dd_pc_ss_{40GB}$	34271.0	29778.0	23986.0	22348.0	17092.0	13997.0	8616.0
$wm_dd_ps_{40GB}$	35007.0	29552.0	27058.0	21728.0	18285.0	18241.0	10618.0
$wm_p_pextb2a2_{40GB}$	35698.0	34365.0	26762.0	26228.0	19184.0	18952.0	12910.0
$wm_p_pextb4a2_{40GB}$	32269.0	32304.0	28707.0	28189.0	20256.0	20607.0	14098.0
wm_ppc_{40GB}	129530.0	128463.0	132225.0	123720.0	156812.0	146338.0	123915.0
$wm_ppc_ss_{40GB}$	157497.0	159163.0	161522.0	155095.0	136040.0	145591.0	123237.0
wm_pps_{40GB}	67595.0	58618.0	55225.0	50521.0	47646.0	41867.0	37395.0
$wt_dd_pc_{40GB}$	37614.0	32440.0	23477.0	19763.0	15454.0	15540.0	10279.0
$wt_dd_pc_ss_{40GB}$	34566.0	28920.0	24619.0	23450.0	15873.0	12347.0	9132.0
$wt_dd_ps_{40GB}$	35361.0	30534.0	27064.0	22377.0	17161.0	16471.0	11666.0
$wt_p_pextb2a3_{40GB}$	31654.0	32873.0	25300.0	25175.0	18360.0	18574.0	12680.0
$wt_p_pextb4a3_{40GB}$	31772.0	32593.0	27840.0	27910.0	20265.0	20034.0	14809.0
wt_ppc_{40GB}	135139.0	130752.0	125093.0	125294.0	155916.0	147629.0	123989.0
$wt_ppc_ss_{40GB}$	165539.0	153463.0	161934.0	145853.0	139166.0	149293.0	121716.0
wt_pps_{40GB}	62662.0	58898.0	54573.0	50769.0	47138.0	41245.0	37310.0
$wm_dd_pc_{30GB}$	27788.0	21559.0	22355.0	19954.0	14821.0	10414.0	7330.0
$wm_dd_pc_ss_{30GB}$	25762.0	21710.0	18254.0	15007.0	12529.0	9114.0	8167.0
$wm_dd_ps_{30GB}$	26795.0	23357.0	19674.0	16081.0	13887.0	12452.0	8389.0
$wm_p_pextb2a2_{30GB}$	25179.0	25522.0	20101.0	19869.0	14886.0	14826.0	9393.0
$wm_p_pextb4a2_{30GB}$	24839.0	24886.0	21950.0	21671.0	15100.0	15191.0	11165.0
wm_ppc_{30GB}	99496.0	95516.0	93757.0	94943.0	113561.0	109307.0	98344.0
$wm_ppc_ss_{30GB}$	121768.0	137663.0	114555.0	111828.0	108458.0	102605.0	100681.0
wm_pps_{30GB}	48795.0	45014.0	44492.0	38186.0	35648.0	30713.0	29102.0
$wt_dd_pc_{30GB}$	26979.0	20993.0	19698.0	17621.0	12896.0	9651.0	5614.0
$wt_dd_pc_ss_{30GB}$	26114.0	21650.0	20502.0	16577.0	13911.0	8590.0	6794.0
$wt_dd_ps_{30GB}$	26657.0	23536.0	19676.0	16527.0	13108.0	10659.0	8597.0
$wt_p_pextb2a3_{30GB}$	25201.0	25125.0	19125.0	19385.0	14237.0	14323.0	9158.0
$wt_p_pextb4a3_{30GB}$	24729.0	24736.0	21216.0	21028.0	14782.0	14924.0	11015.0
wt_ppc_{30GB}	99164.0	146028.0	94016.0	95132.0	107593.0	103567.0	93699.0
$wt_ppc_ss_{30GB}$	119380.0	119192.0	107613.0	110821.0	109353.0	106139.0	91225.0
wt_pps_{30GB}	48640.0	44230.0	43798.0	38588.0	35374.0	30807.0	28644.0

Algorithm	8	7	6	5	4	3	2
$wm_dd_pc_{20GB}$	16323.0	17960.0	15261.0	12144.0	9358.0	7998.0	5132.0
$wm_dd_pc_ss_{20GB}$	17395.0	14852.0	13504.0	10487.0	8384.0	6181.0	4285.0
$wm_dd_ps_{20GB}$	17608.0	15184.0	13098.0	10813.0	9036.0	7033.0	5301.0
$wm_p_pextb2a2_{20GB}$	16701.0	16918.0	13057.0	13272.0	9736.0	9652.0	5887.0
$wm_p_pextb4a2_{20GB}$	16277.0	15894.0	13819.0	13939.0	10105.0	9864.0	7074.0
wm_ppc_{20GB}	66063.0	64849.0	61854.0	67228.0	74680.0	74110.0	62350.0
$wm_ppc_ss_{20GB}$	84234.0	86476.0	81693.0	78307.0	71863.0	74261.0	61621.0
wm_pps_{20GB}	32426.0	31130.0	29011.0	25910.0	23389.0	20422.0	18610.0
$wt_dd_pc_{20GB}$	16109.0	13991.0	13386.0	9984.0	7713.0	5881.0	3714.0
$wt_dd_pc_ss_{20GB}$	16963.0	14381.0	13026.0	10360.0	8220.0	6212.0	4297.0
$wt_dd_ps_{20GB}$	17100.0	14934.0	13712.0	10928.0	8904.0	6957.0	5305.0
$wt_p_pextb2a3_{20GB}$	16666.0	16816.0	12703.0	12762.0	9533.0	9417.0	5894.0
$wt_p_pextb4a3_{20GB}$	15676.0	15672.0	13740.0	13881.0	9724.0	9876.0	7508.0
wt_ppc_{20GB}	66016.0	63234.0	63394.0	63039.0	76682.0	72704.0	63208.0
$wt_ppc_ss_{20GB}$	80416.0	80725.0	86005.0	77757.0	72652.0	75556.0	61332.0
wt_pps_{20GB}	32139.0	31248.0	27645.0	24969.0	22854.0	20387.0	18554.0
$wm_dd_pc_{10GB}$	9637.0	7870.0	7560.0	6476.0	3794.0	3663.0	2531.0
$wm_dd_pc_ss_{10GB}$	8590.0	7362.0	6125.0	5036.0	4563.0	3199.0	2158.0
$wm_dd_ps_{10GB}$	8742.0	7727.0	6418.0	5350.0	4616.0	3558.0	2661.0
$wm_p_pextb2a2_{10GB}$	8299.0	8338.0	6481.0	6530.0	4860.0	4757.0	2997.0
$wm_p_pextb4a2_{10GB}$	8034.0	7771.0	6990.0	6844.0	4925.0	4840.0	3613.0
wm_ppc_{10GB}	48098.0	32704.0	30723.0	31027.0	37205.0	33221.0	32483.0
$wm_ppc_ss_{10GB}$	42752.0	44302.0	39049.0	37778.0	36396.0	37911.0	31251.0
wm_pps_{10GB}	16197.0	14844.0	13665.0	12560.0	11735.0	10644.0	9552.0
$wt_dd_pc_{10GB}$	8482.0	7054.0	5937.0	5518.0	4242.0	2804.0	1762.0
$wt_dd_pc_ss_{10GB}$	8835.0	7309.0	6139.0	5020.0	4054.0	3022.0	2617.0
$wt_dd_ps_{10GB}$	8553.0	7653.0	6372.0	5309.0	4584.0	3718.0	2740.0
$wt_p_pextb2a3_{10GB}$	8308.0	8191.0	6274.0	6447.0	4661.0	4532.0	3019.0
$wt_p_pextb4a3_{10GB}$	7871.0	7724.0	6739.0	6805.0	4945.0	4897.0	3697.0
wt_ppc_{10GB}	46207.0	32761.0	31177.0	31555.0	50586.0	36927.0	31412.0
$wt_ppc_ss_{10GB}$	44037.0	38229.0	37352.0	35138.0	35923.0	37930.0	30713.0
wt_pps_{10GB}	16234.0	14958.0	13977.0	12632.0	11782.0	10256.0	9491.0
$wm_dd_pc_{1GB}$	707.0	682.0	493.0	407.0	383.0	280.0	179.0
$wm_dd_pc_ss_{1GB}$	886.0	741.0	533.0	511.0	453.0	243.0	186.0
$wm_dd_ps_{1GB}$	709.0	652.0	561.0	415.0	451.0	269.0	139.0
$wm_p_pextb2a2_{1GB}$	682.0	710.0	546.0	532.0	377.0	368.0	220.0
$wm_p_pextb4a2_{1GB}$	630.0	628.0	545.0	544.0	376.0	386.0	261.0
wm_ppc_{1GB}	3312.0	3198.0	3063.0	3135.0	3800.0	3865.0	3082.0
$wm_ppc_ss_{1GB}$	5807.0	4011.0	3876.0	3697.0	3417.0	3779.0	3073.0
wm_pps_{1GB}	1697.0	1577.0	1410.0	1269.0	1191.0	1051.0	951.0
$wt_dd_pc_{1GB}$	736.0	700.0	505.0	457.0	382.0	288.0	181.0
$wt_dd_pc_ss_{1GB}$	891.0	725.0	779.0	564.0	404.0	318.0	184.0
$wt_dd_ps_{1GB}$	705.0	763.0	468.0	382.0	421.0	336.0	217.0
$wt_p_pextb2a3_{1GB}$	673.0	680.0	505.0	513.0	373.0	363.0	220.0
$wt_p_pextb4a3_{1GB}$	620.0	646.0	522.0	522.0	403.0	360.0	266.0
wt_ppc_{1GB}	3259.0	3305.0	3157.0	3239.0	3912.0	3851.0	3128.0
$wt_ppc_ss_{1GB}$	5369.0	3907.0	4282.0	3432.0	3527.0	3612.0	3039.0
wt_pps_{1GB}	1666.0	1542.0	1380.0	1267.0	1191.0	1028.0	957.0

A.2. Histograms of example texts