Bit-Parallel (Compressed) Wavelet Tree Construction

Patrick Dinklage
technische universität dortmund

Johannes Fischer
technische universität dortmund

Florian Kurpicz
KIT Karlsruher Institut für Technologie

Jan-Philipp Tarnowski
crownpeak
technische universität dortmund

Wavelet Trees

wavelet_tree

Char	Code
_	000
a	001
e	010
l	011
r	100
t	101
v	110
w	111

Wavelet Trees

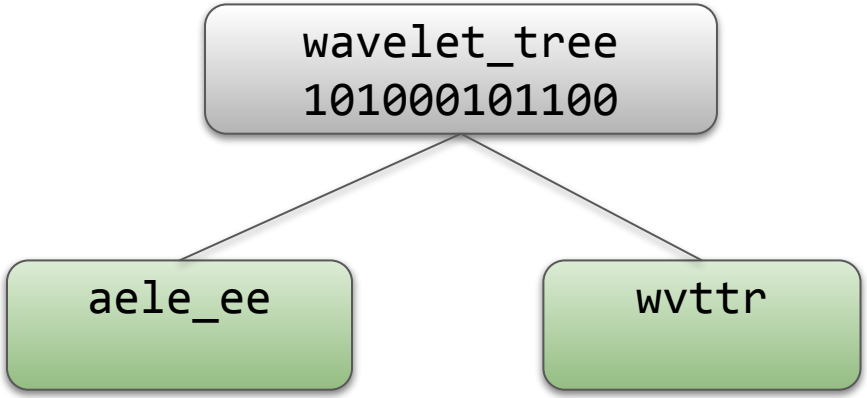
level 1
(most significant bits)

wavelet_tree
101000101100

Char	Code
_	000
a	001
e	010
l	011
r	100
t	101
v	110
w	111

Wavelet Trees

level 1
(most significant bits)



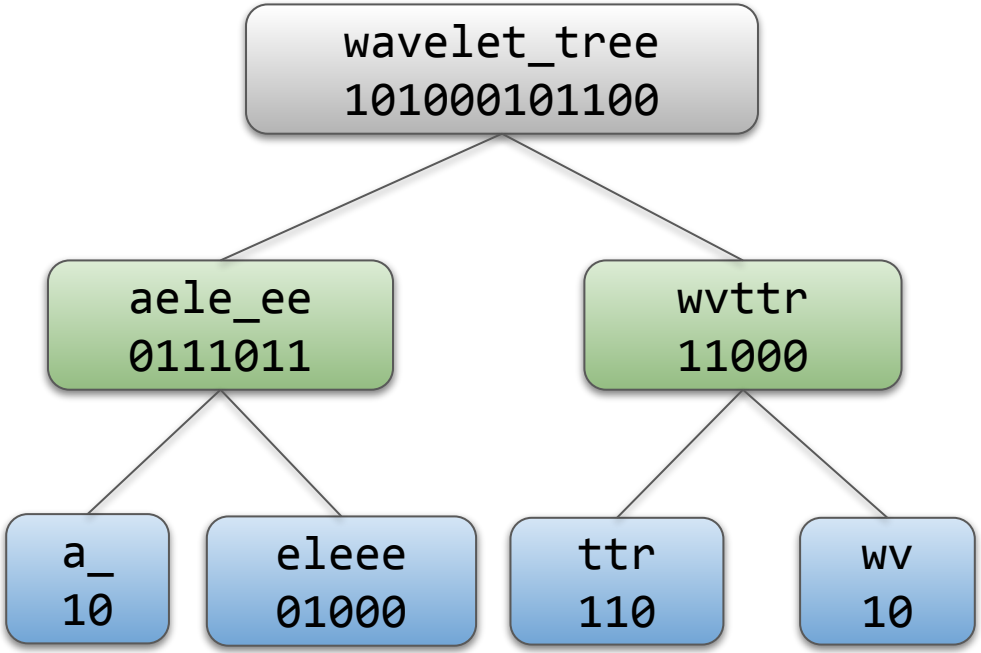
Char	Code
_	000
a	001
e	010
l	011
r	100
t	101
v	110
w	111

Wavelet Trees

level 1
(most significant bits)

level 2
(second bits)

level 3
(third bits)



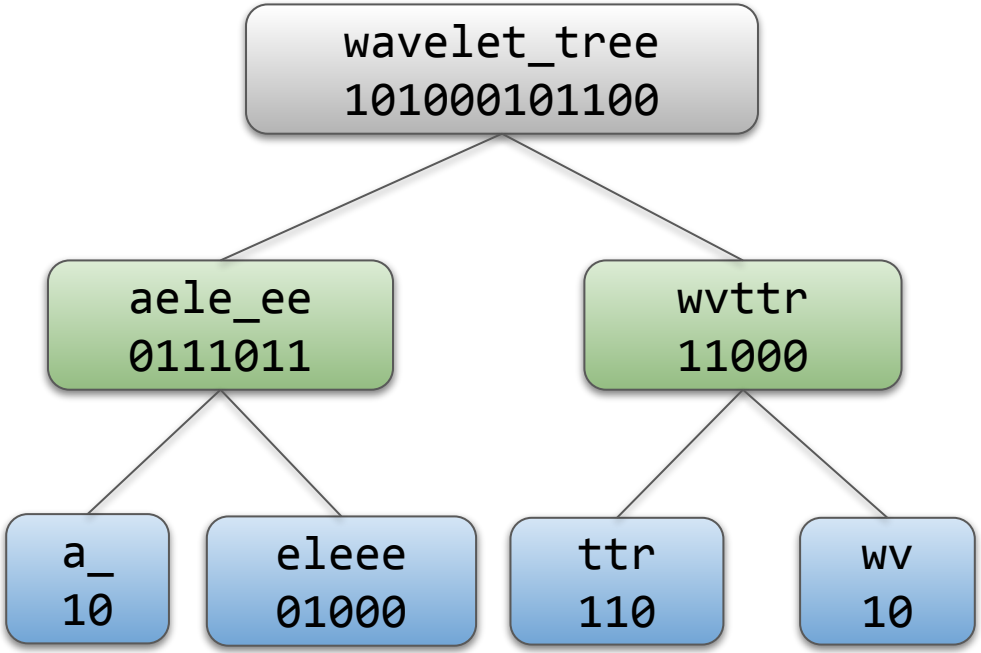
Char	Code
_	000
a	001
e	010
l	011
r	100
t	101
v	110
w	111

Wavelet Trees

level 1
(most significant bits)

level 2
(second bits)

level 3
(third bits)



Char	Code
_	000
a	001
e	010
l	011
r	100
t	101
v	110
w	111

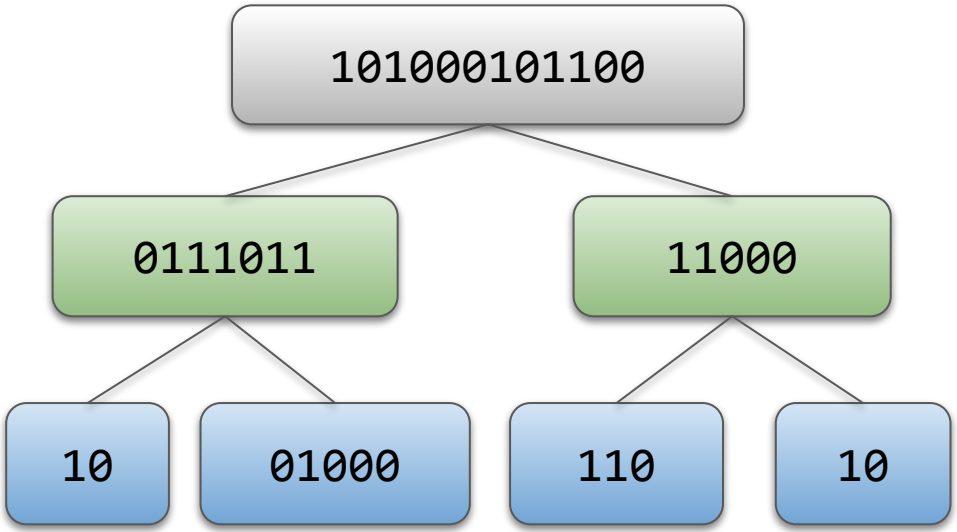
➔ $\lceil lg\ \sigma \rceil$ levels, n bits per level

Wavelet Trees

level 1
(most significant bits)

level 2
(second bits)

level 3
(third bits)



Char	Code
_	000
a	001
e	010
l	011
r	100
t	101
v	110
w	111

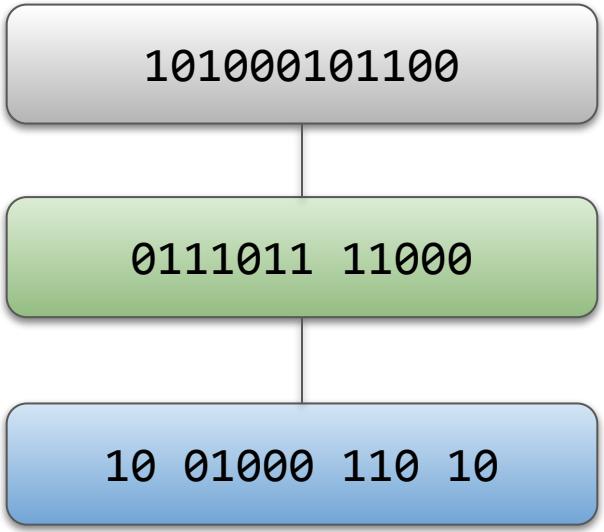
➔ we only store the bits, text remains decodable

Wavelet Trees

level 1
(most significant bits)

level 2
(second bits)

level 3
(third bits)



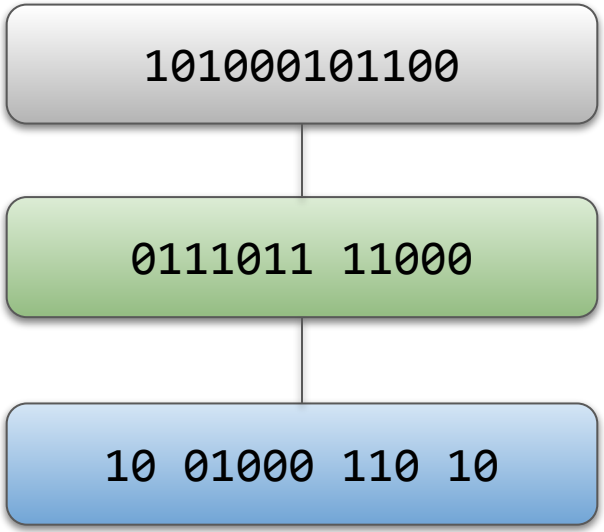
Char	Code
_	000
a	001
e	010
l	011
r	100
t	101
v	110
w	111

Wavelet Trees

level 1
(most significant bits)

level 2
(second bits)

level 3
(third bits)



Char	Code
_	000
a	001
e	010
l	011
r	100
t	101
v	110
w	111

➔ **levelwise** (pointerless) representation fits in $\lceil lg\ \sigma \rceil (n+o(n))$ bits

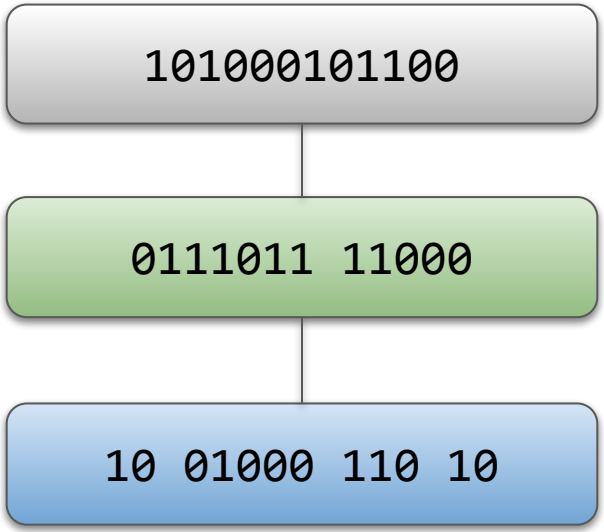
(tree structure is retained implicity)

Wavelet Trees

level 1
(most significant bits)

level 2
(second bits)

level 3
(third bits)



Char	Code
_	000
a	001
e	010
l	011
r	100
t	101
v	110
w	111

➔ **levelwise** (pointerless) representation fits in $\lceil lg\ \sigma \rceil (n+o(n))$ bits

(tree structure is retained implicity)

➔ applications in compressed text indexing (e.g., FM-Index)

Wavelet Tree Construction

Construction algorithm	Time bound (seq.)
text book prefix counting [D. et al., 2022]	$\mathcal{O}(n \lg \sigma)$

Wavelet Tree Construction

Construction algorithm	Time bound (seq.)
text book prefix counting [D. et al., 2022]	$\mathcal{O}(n \lg \sigma)$
[Babenko et al., 2014] [Munro et al., 2014] [Kaneta, 2018]	$\mathcal{O}(n \lg \sigma / \sqrt{\lg n})$

Wavelet Tree Construction

Construction algorithm	Time bound (seq.)
text book prefix counting [D. et al., 2022]	$\mathcal{O}(n \lg \sigma)$
[Babenko et al., 2014] [Munro et al., 2014] [Kaneta, 2018]	$\mathcal{O}(n \lg \sigma / \sqrt{\lg n})$

This work:

1. Does Kaneta's algorithm scale with the register size (AVX-512)?

Wavelet Tree Construction

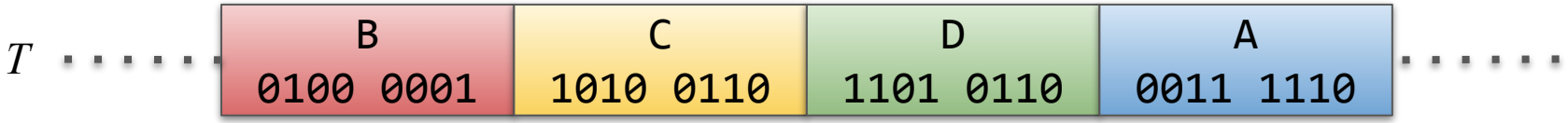
Construction algorithm	Time bound (seq.)
text book prefix counting [D. et al., 2022]	$\mathcal{O}(n \lg \sigma)$
[Babenko et al., 2014] [Munro et al., 2014] [Kaneta, 2018]	$\mathcal{O}(n \lg \sigma / \sqrt{\lg n})$

This work:

1. Does Kaneta's algorithm scale with the register size (AVX-512)?
2. Can it be adapted to build Huffman-shaped (compressed) wavelet trees?

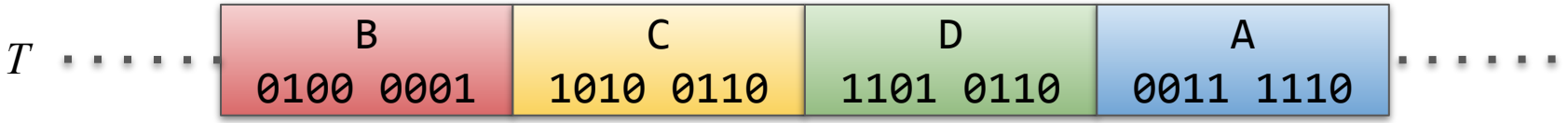
Fast Levelwise Construction

(assuming a byte alphabet of size $\sigma = 256$)

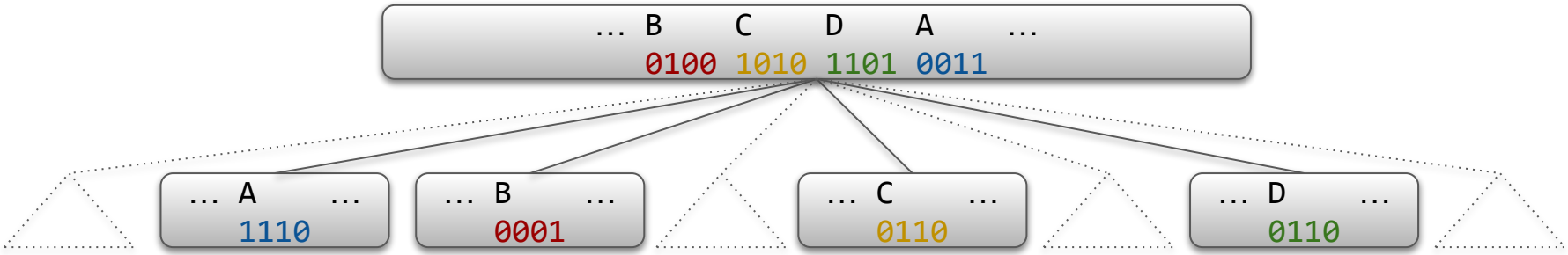


Fast Levelwise Construction

(assuming a byte alphabet of size $\sigma = 256$)



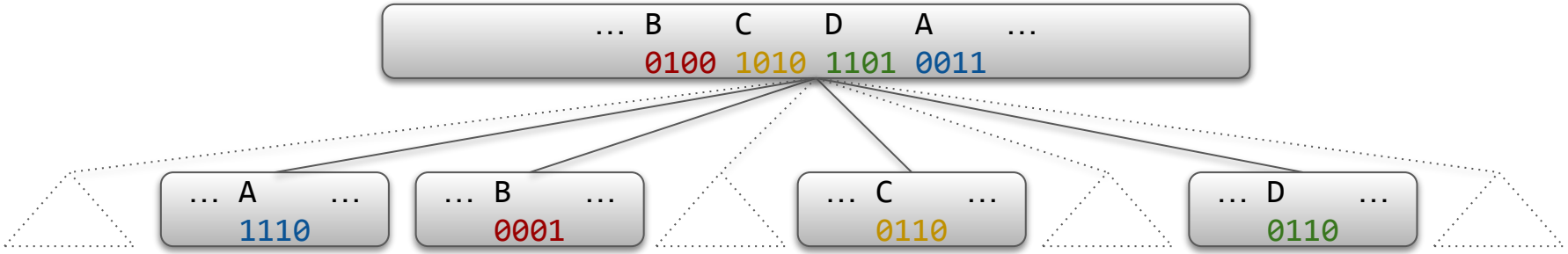
➔ we build a 2^τ-ary levelwise wavelet tree (ex.: $\tau = 4$)



➔ $\lceil lg\ \sigma \rceil / \tau$ levels

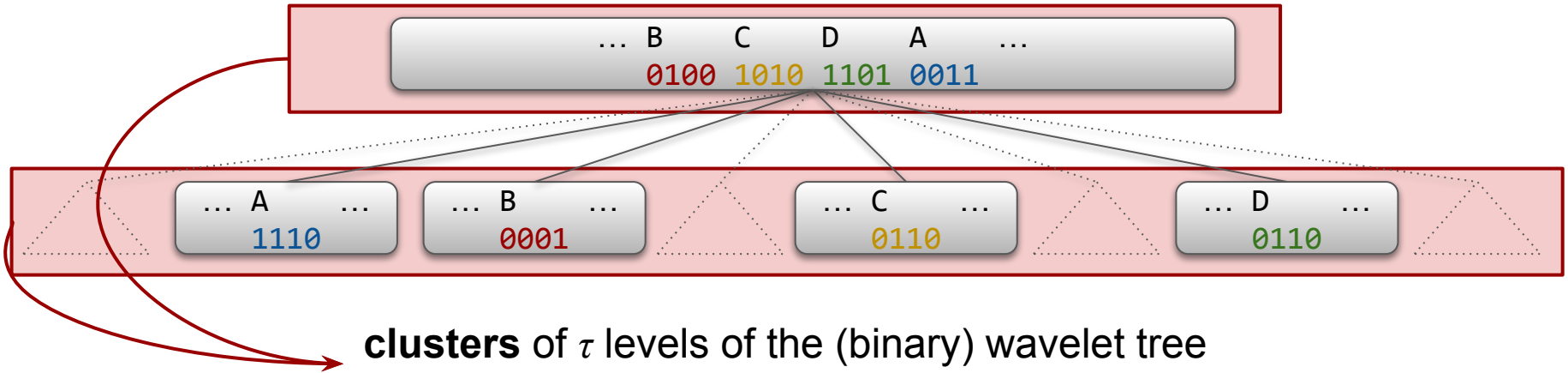
Fast Levelwise Construction

(we build a 2^{τ}-ary levelwise wavelet tree (ex.: $\tau = 4$))



Fast Levelwise Construction

(we build a 2^τ-ary levelwise wavelet tree (ex.: $\tau = 4$))

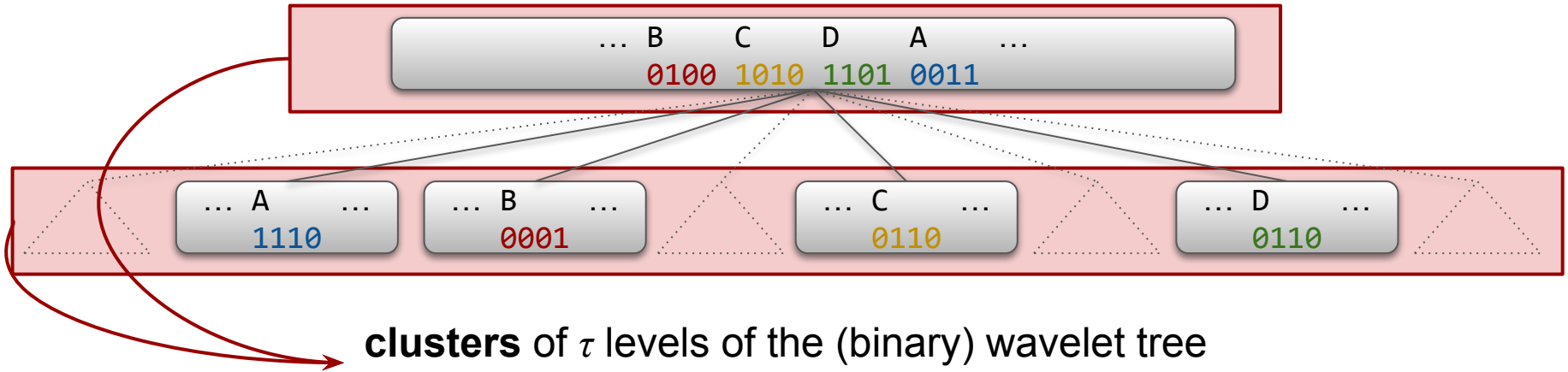


clusters of τ levels of the (binary) wavelet tree

(array of τ-bit blocks)

Fast Levelwise Construction

(we build a 2^τ-ary levelwise wavelet tree (ex.: $\tau = 4$))

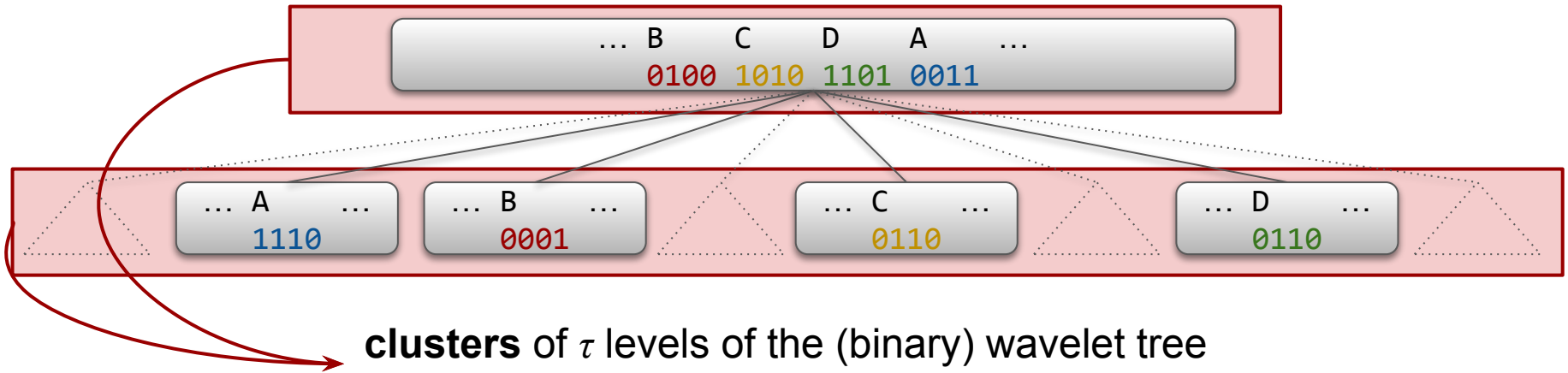


clusters of τ levels of the (binary) wavelet tree
(array of τ-bit blocks)

➔ **Grand strategy**: expand a cluster to τ bit vectors in time $O(n)$

Fast Levelwise Construction

(we build a 2^τ-ary levelwise wavelet tree (ex.: $\tau = 4$))



clusters of τ levels of the (binary) wavelet tree
(array of τ-bit blocks)

➔ **Grand strategy**: expand a cluster to τ bit vectors in time $O(n)$

➔ then, for all $\lceil lg\ \sigma \rceil / \tau$ clusters, the total construction time becomes $O(n\ lg\ \sigma/\tau)$

$(\tau := \sqrt{\lg n})$

Fast Levelwise Construction

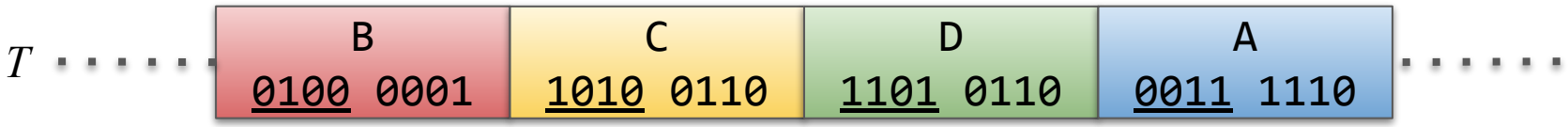
our steps **for each cluster**:

#	Step	Running time
1	Cluster Extraction	$O(n)$
2	τ passes	$O(n\tau^2/\lg n) = O(n)$
2a	- Bit Extraction	- $O(n\tau/\lg n)$
2b	- List Splitting	- $O(n\tau/\lg n)$
3	Text Reshuffling	$O(n)$

$(\tau := \sqrt{\lg n})$

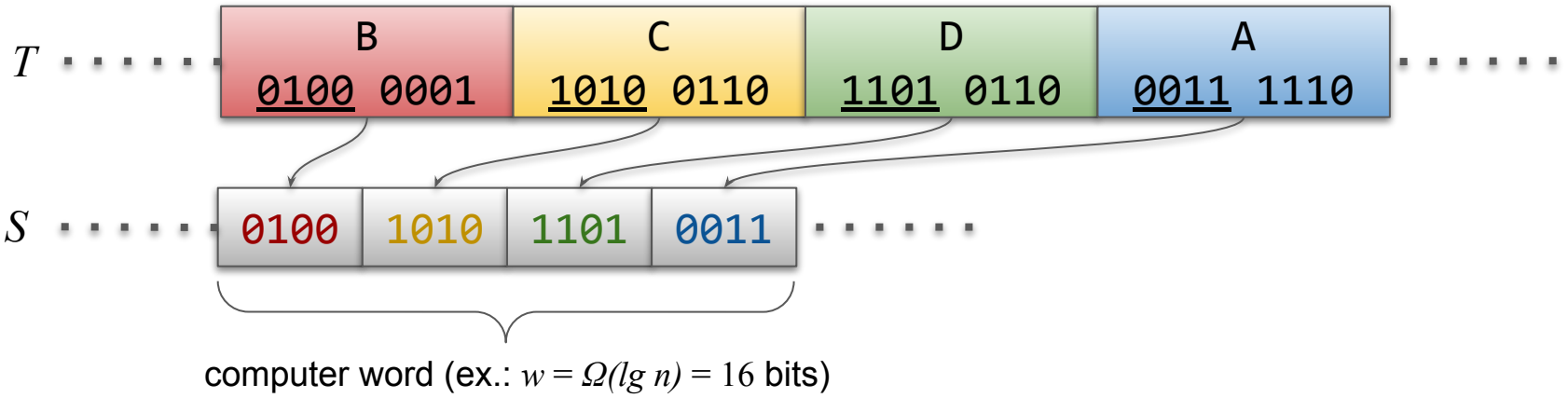
Cluster Extraction

➔ extract the relevant block of τ bits from each character to a **word-packed list** S



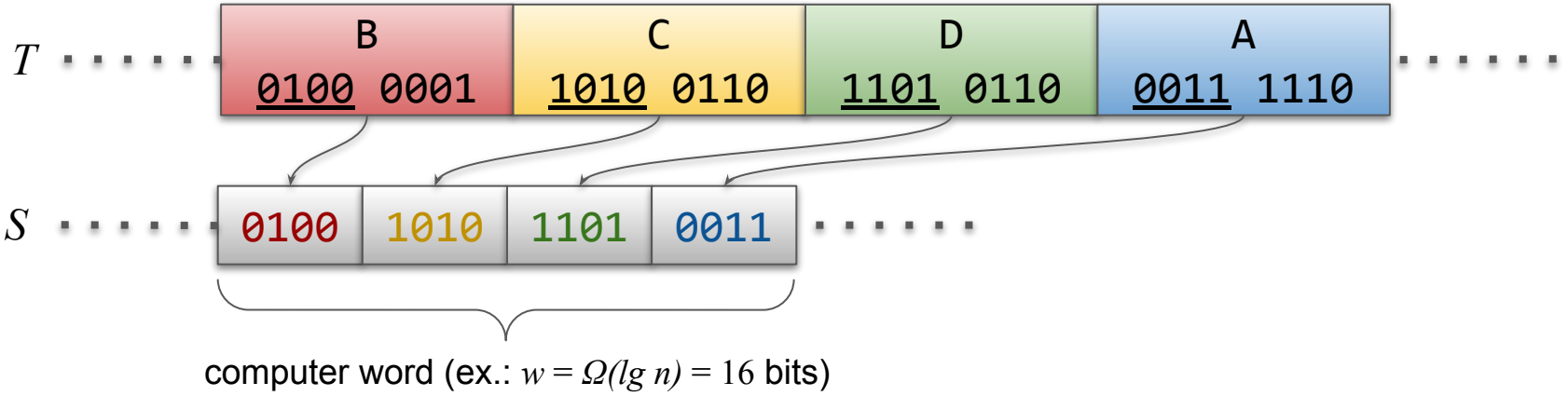
Cluster Extraction

➔ extract the relevant block of τ bits from each character to a **word-packed list** S



Cluster Extraction

➔ extract the relevant block of τ bits from each character to a **word-packed list** S



➔ left-to-right scan of T in time $O(n)$ ✓

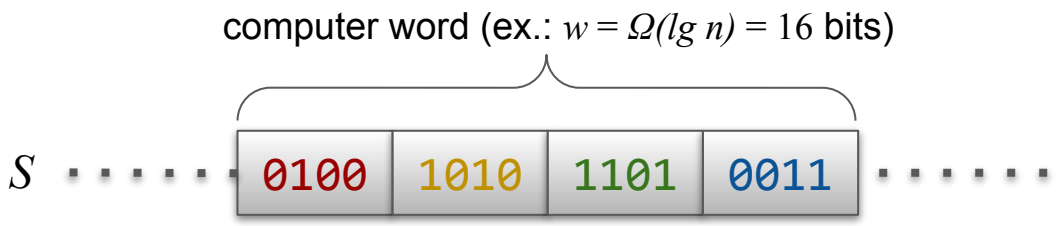
Fast Levelwise Construction

steps for each cluster:

#	Step	Running time
1	Cluster Extraction	$O(n)$
2	τ passes	$O(n\tau^2/\lg n) = O(n)$
2a	- Bit Extraction	- $O(n\tau/\lg n)$
2b	- List Splitting	- $O(n\tau/\lg n)$
3	Text Reshuffling	$O(n)$

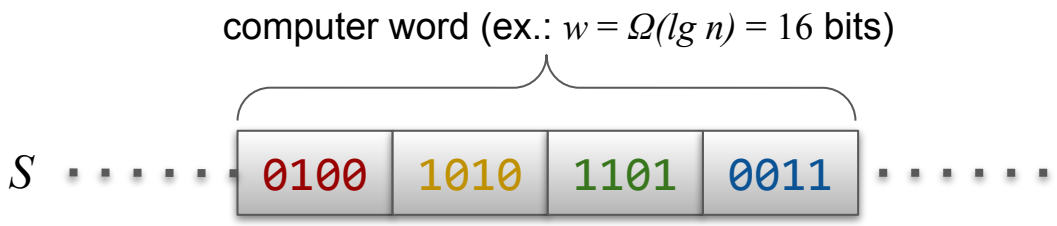
$(\tau := \sqrt{\lg n})$

Word Packing



➔ we pack w/τ blocks into a word / S consists of $n\tau/w$ words

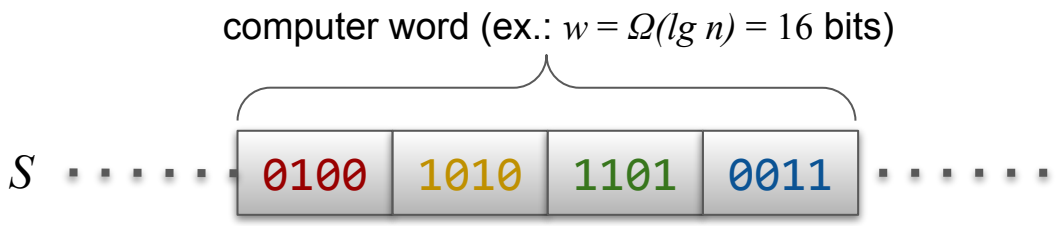
Word Packing



➔ we pack w/τ blocks into a word / S consists of $n\tau/w$ words

➔ if processing one word takes constant time,
then processing S takes time $O(n\tau/w)$

Word Packing

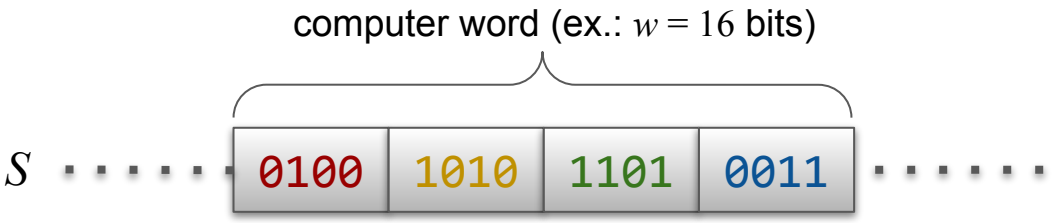


➔ we pack w/τ blocks into a word / S consists of $n\tau/w$ words

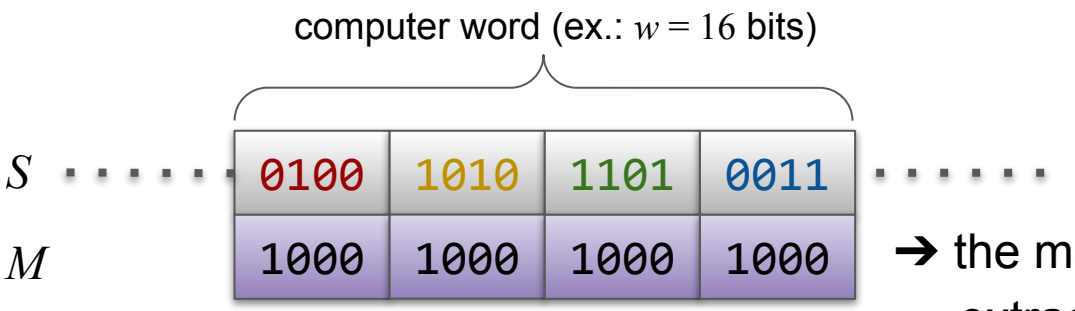
➔ if processing one word takes constant time,
then processing S takes time $O(n\tau/w)$

➔ we will do τ **passes** over S, each pass taking time $O(n\tau/w) = O(n\tau/\lg n)$ ✓

Bit Extraction

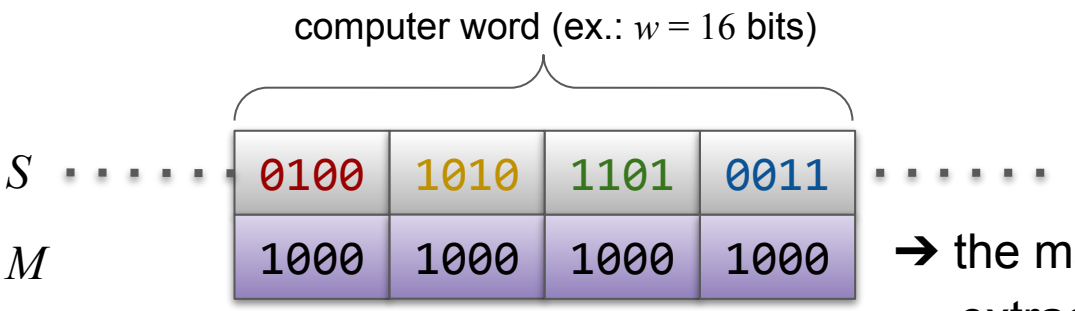


Bit Extraction



➔ the mask M marks what bits to extract from a word

Bit Extraction

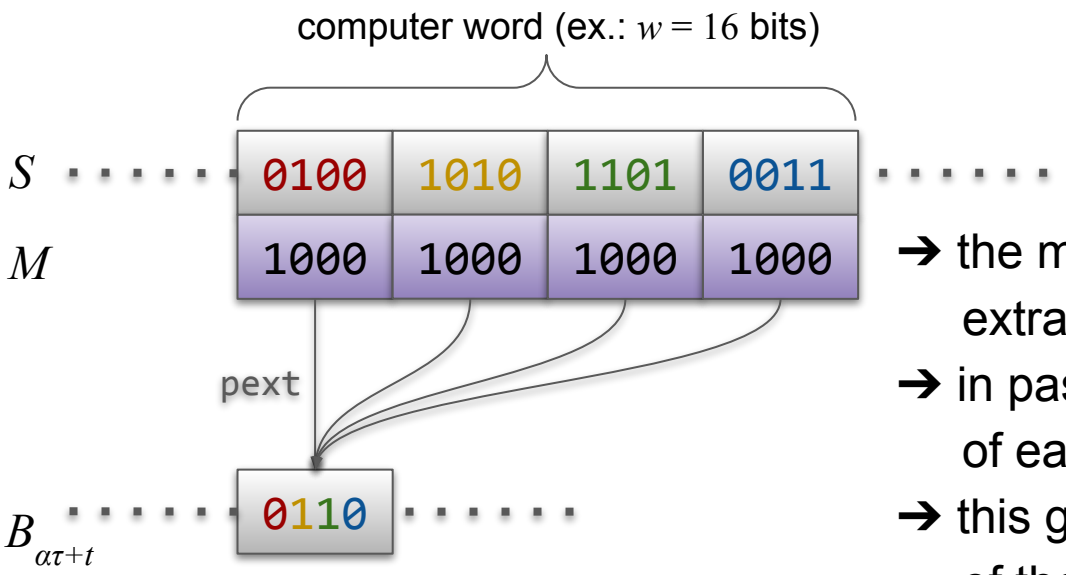


➔ the mask M marks what bits to extract from a word

➔ in pass t, we set the t-th bit of each block

➔ this gives us the bits for level $\alpha\tau+t$ of the wavelet tree

Bit Extraction

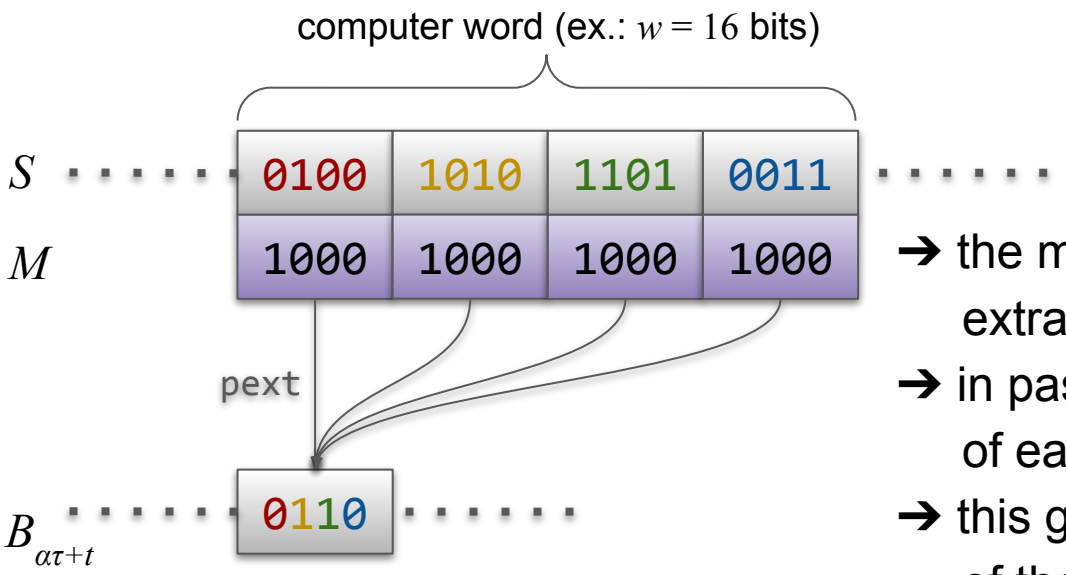


➔ the mask M marks what bits to extract from a word

➔ in pass t, we set the t-th bit of each block

➔ this gives us the bits for level $\alpha\tau+t$ of the wavelet tree

Bit Extraction



➔ the mask M marks what bits to extract from a word

➔ in pass t, we set the t-th bit of each block

➔ this gives us the bits for level $\alpha\tau+t$ of the wavelet tree

➔ using lookup tables, this can be done in constant time per word

(in practice, we use the `pext` CPU instruction*)

Fast Levelwise Construction

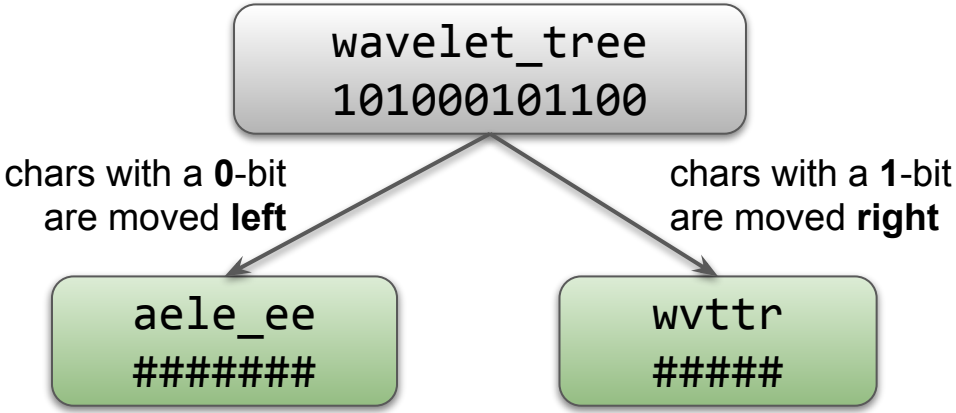
steps for each cluster:

#	Step	Running time
✓ 1	Cluster Extraction	$O(n)$
2	τ passes	$O(n\tau^2/\lg n) = O(n)$
✓ 2a	- Bit Extraction	- $O(n\tau/\lg n)$
2b	- List Splitting	- $O(n\tau/\lg n)$
3	Text Reshuffling	$O(n)$

$(\tau := \sqrt{\lg n})$

List Splitting

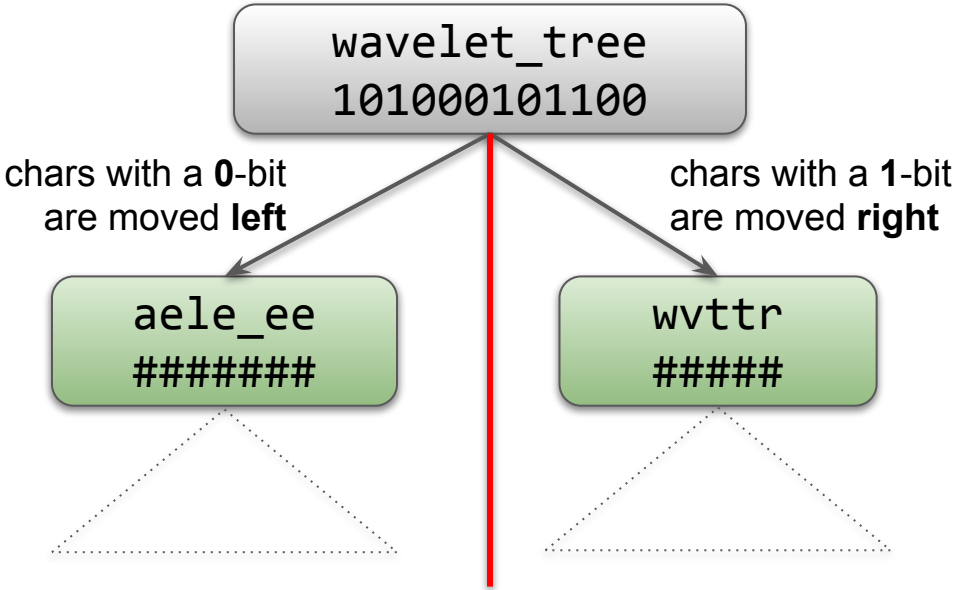
recall how the text is “split” at a wavelet tree node



➔ we need to simulate this on S after every pass

List Splitting

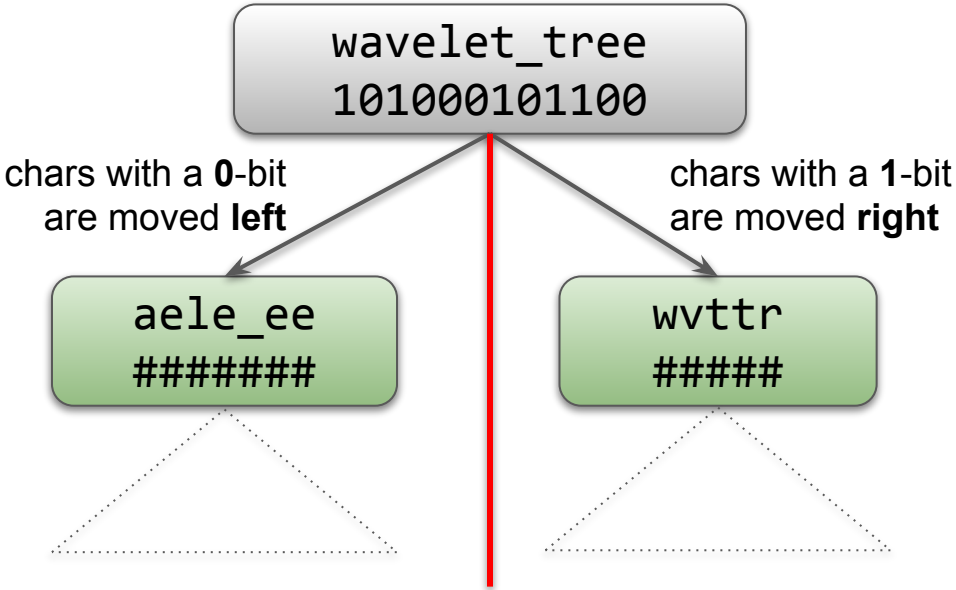
recall how the text is “split” at a wavelet tree node



➔ the **border** position equals the number of chars with a 0-bit

List Splitting

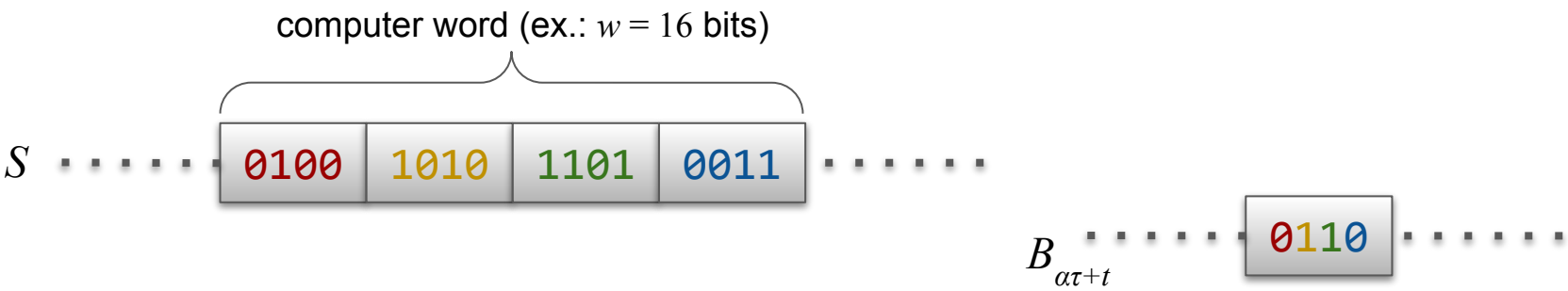
recall how the text is “split” at a wavelet tree node



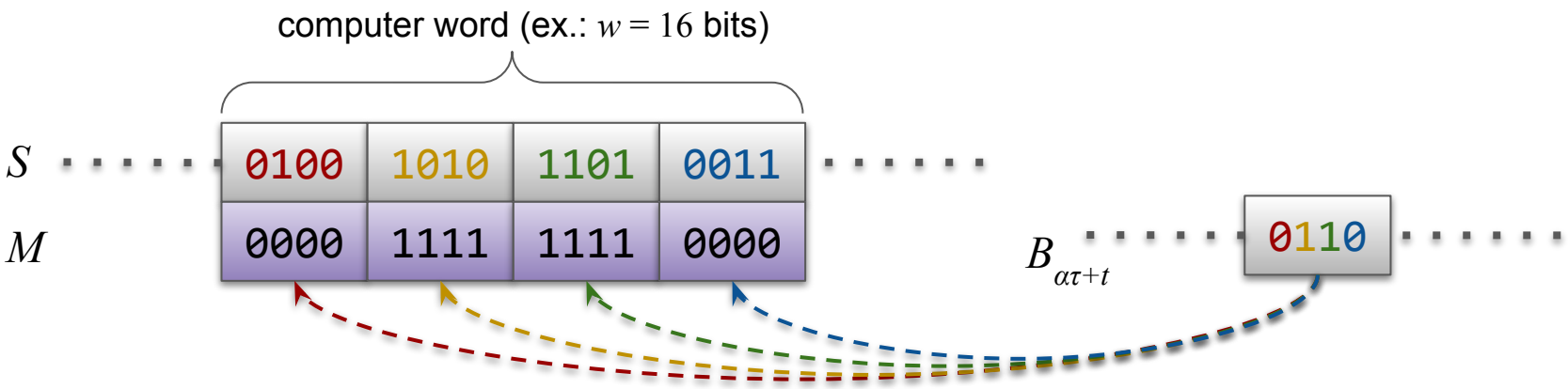
➔ the **border** position equals the number of chars with a 0-bit

➔ important: counting 0-bits must be done in constant time per word! (popcount)

List Splitting



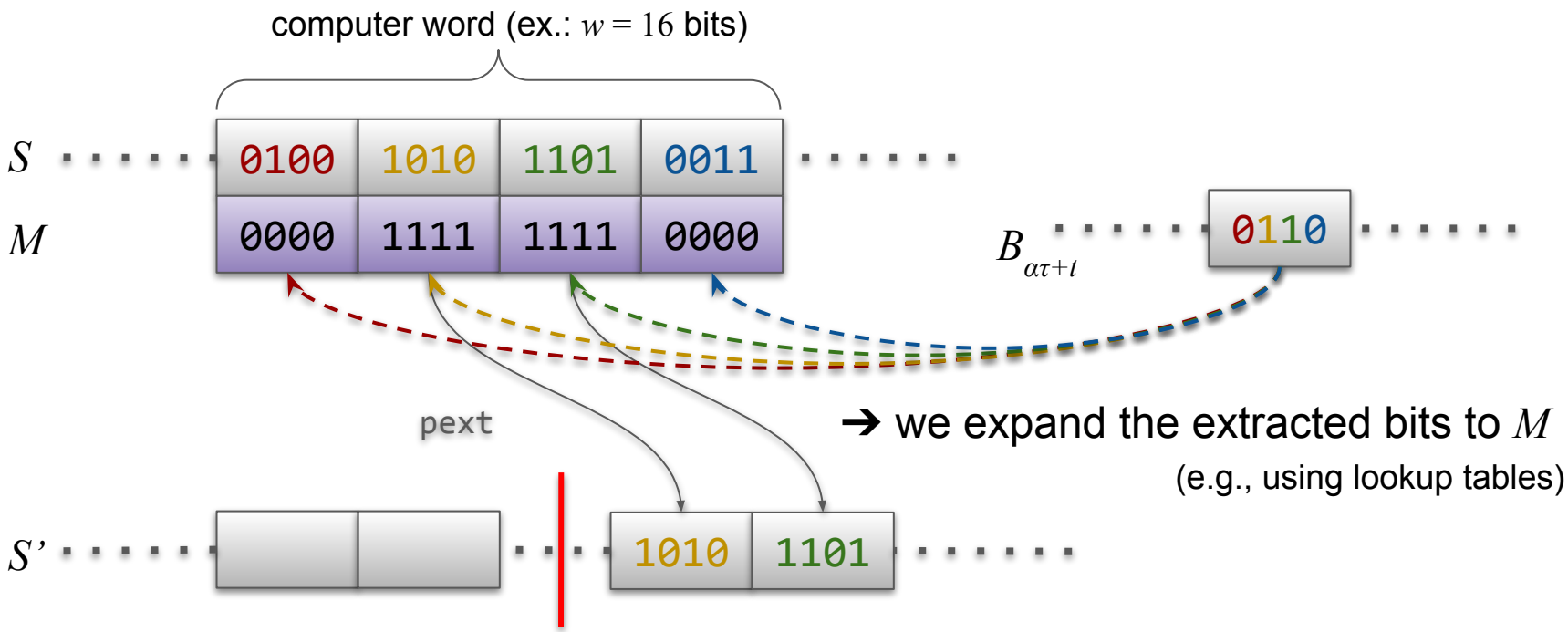
List Splitting



➔ we expand the extracted bits to M

(e.g., using lookup tables)

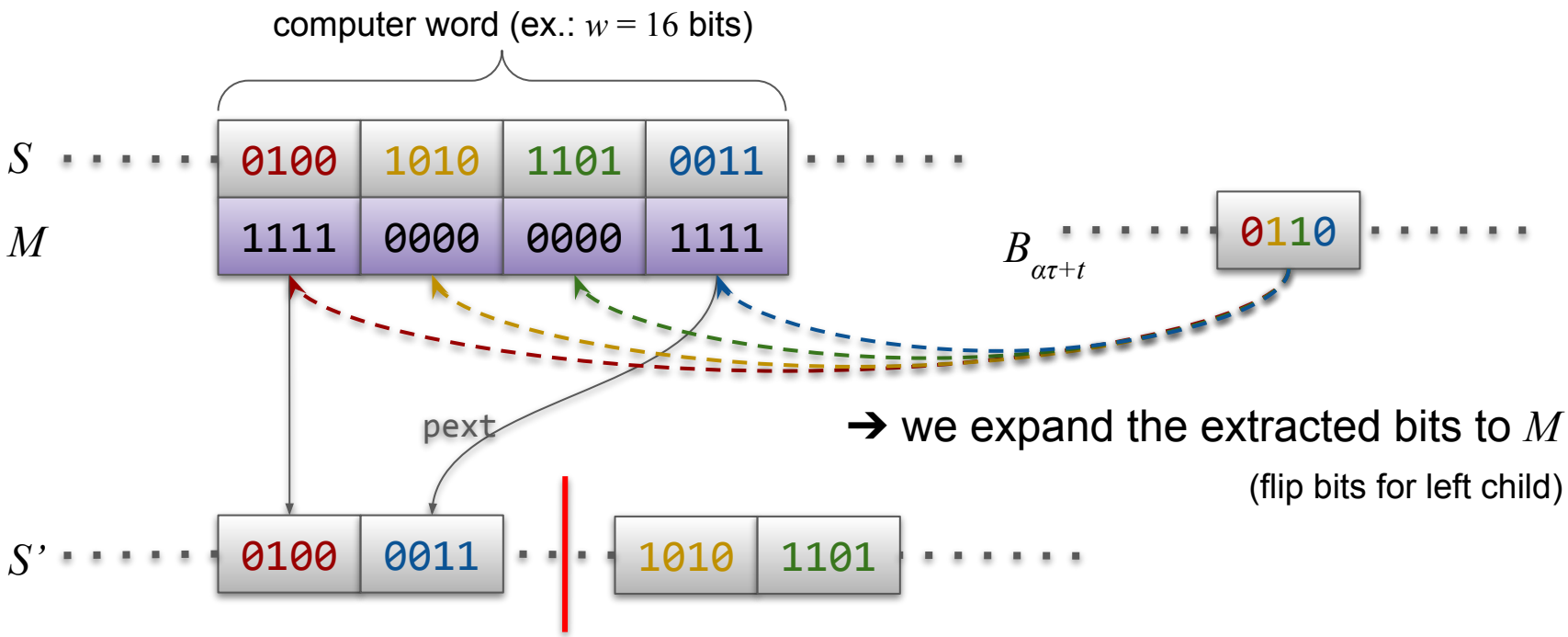
List Splitting



➔ we expand the extracted bits to M

(e.g., using lookup tables)

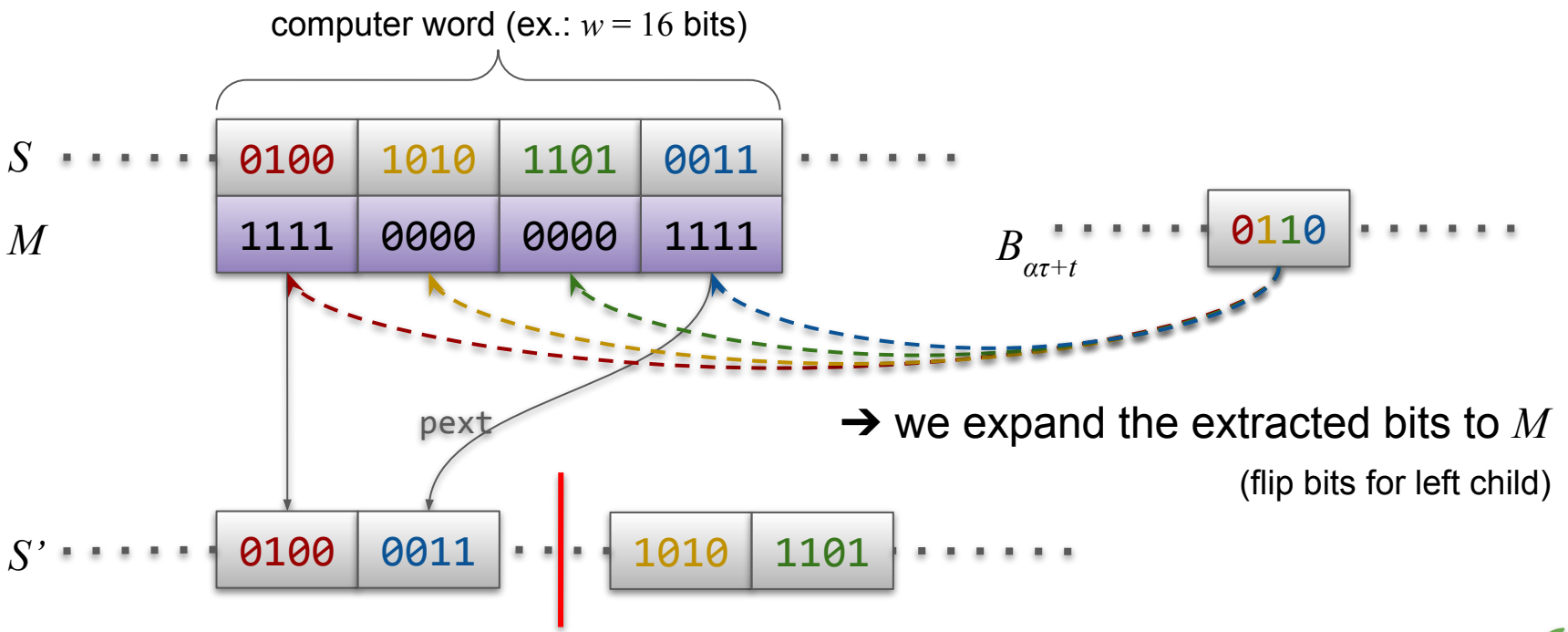
List Splitting



➔ we expand the extracted bits to M

(flip bits for left child)

List Splitting



➔ we expand the extracted bits to M

(flip bits for left child)

➔ using `pext*` twice, we do the desired splitting in constant time per word ✓

Fast Levelwise Construction

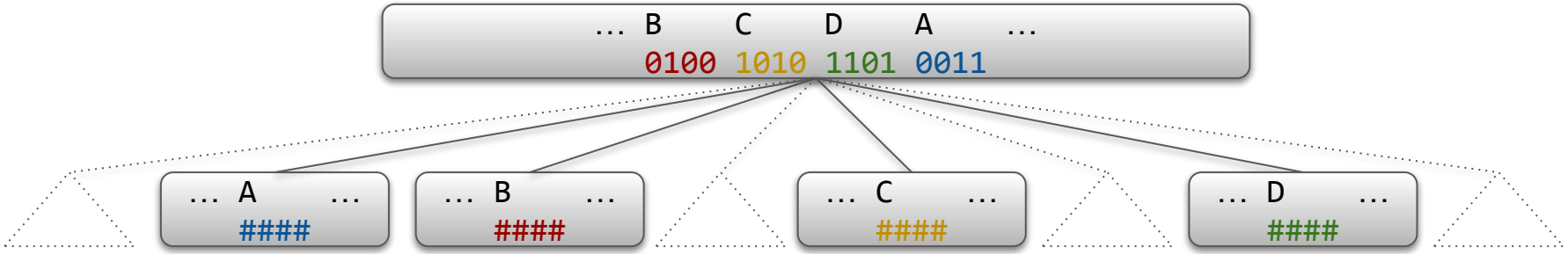
steps for each cluster:

#	Step	Running time
1	Cluster Extraction	$O(n)$
2	τ passes	$O(n\tau^2/\lg n) = O(n)$
2a	- Bit Extraction	- $O(n\tau/\lg n)$
2b	- List Splitting	- $O(n\tau/\lg n)$
3	Text Reshuffling	$O(n)$

$(\tau := \sqrt{\lg n})$

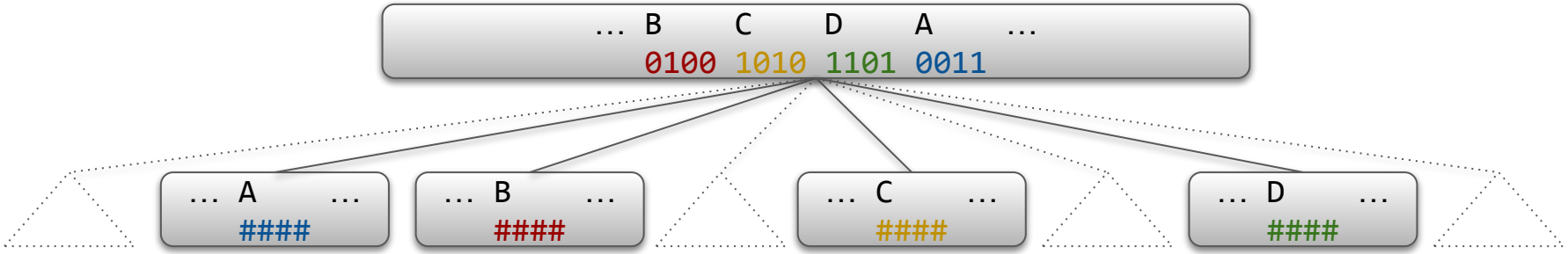
Text Reshuffling

recall how the text is “split” at a generalized wavelet tree node



Text Reshuffling

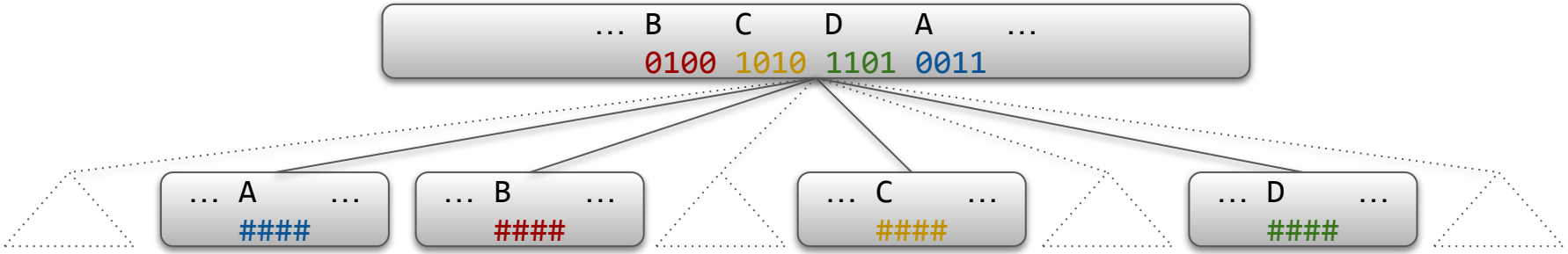
recall how the text is “split” at a generalized wavelet tree node



➔ chars with bit pattern $(v)_2$ are moved to the v-th child node

Text Reshuffling

recall how the text is “split” at a generalized wavelet tree node



➔ chars with bit pattern $(v)_2$ are moved to the v-th child node

➔ we need to simulate this on T after finishing a cluster of τ levels

➔ this is essentially **stable counting sort**,
and the WT node borders are already known!

Fast Levelwise Construction

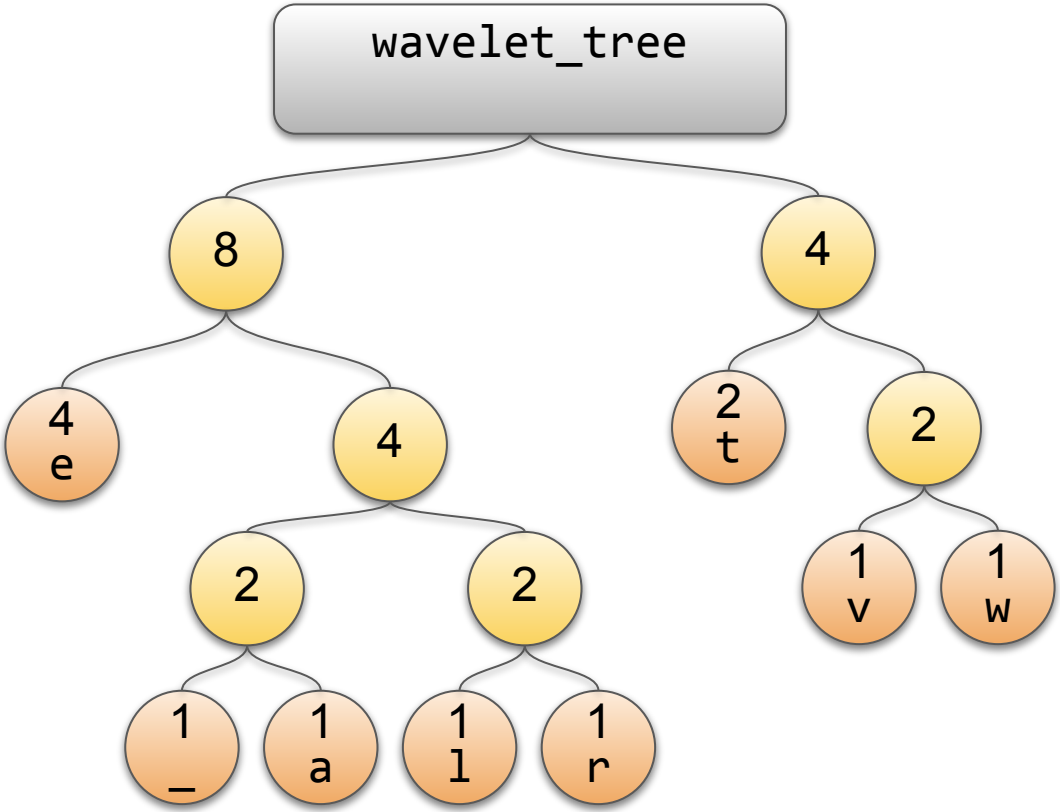
steps for each cluster:

#	Step	Running time
1	Cluster Extraction	$O(n)$
2	τ passes	$O(n\tau^2/\lg n) = O(n)$
2a	- Bit Extraction	- $O(n\tau/\lg n)$
2b	- List Splitting	- $O(n\tau/\lg n)$
3	Text Reshuffling	$O(n)$

$(\tau := \sqrt{\lg n})$

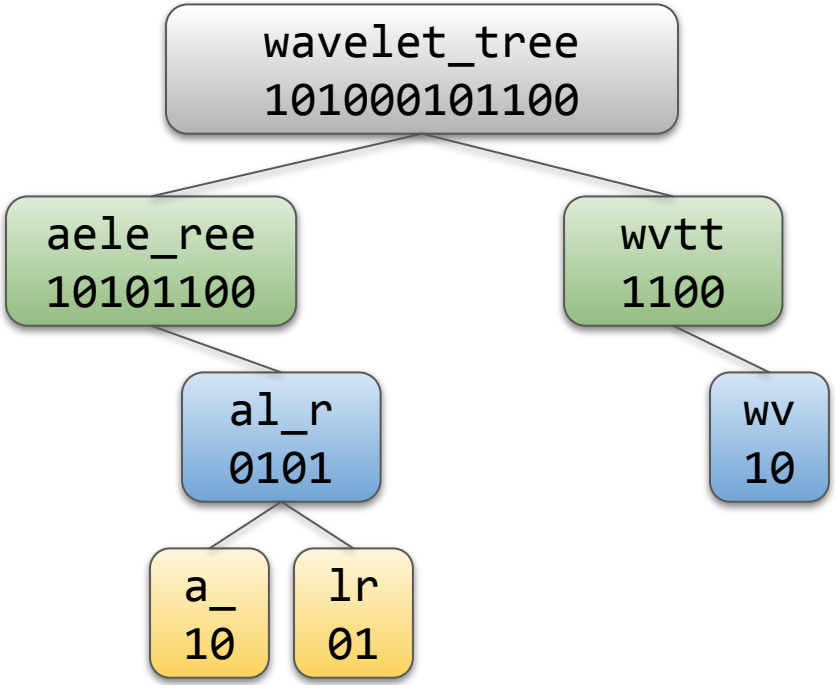
➔ for all $\lceil \lg \sigma \rceil / \tau$ clusters, the total construction time becomes $O(n \lg \sigma/\tau)$

Huffman-Shaped Wavelet Trees



Char	Code
_	0100
a	0101
e	00--
l	0110
r	0111
t	10--
v	110-
w	111-

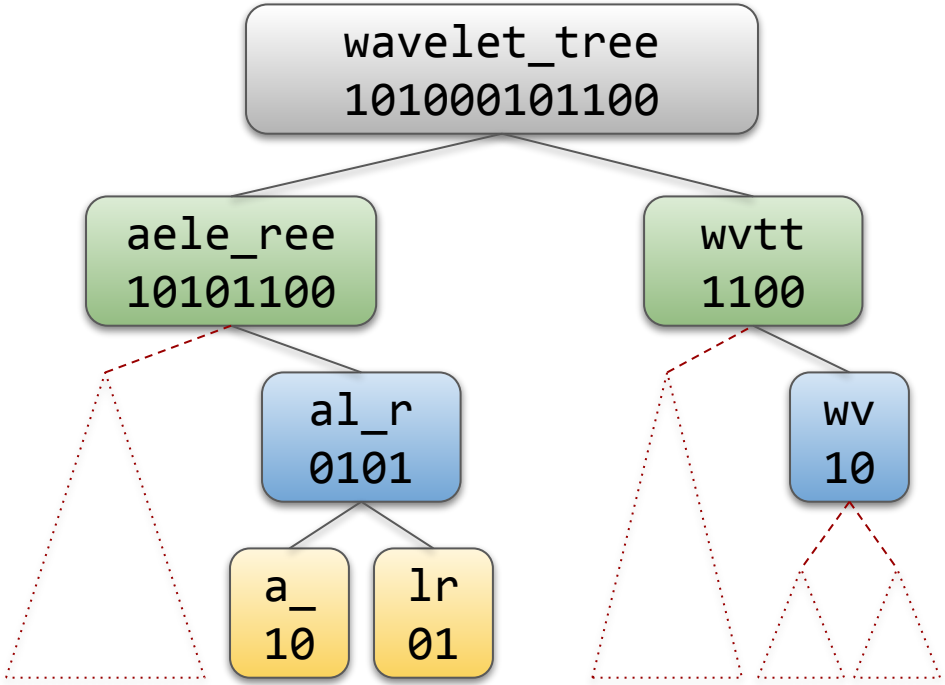
Huffman-Shaped Wavelet Trees



Char	Code
_	0100
a	0101
e	00--
l	0110
r	0111
t	10--
v	110-
w	111-

➔ build wavelet tree according to Huffman codes

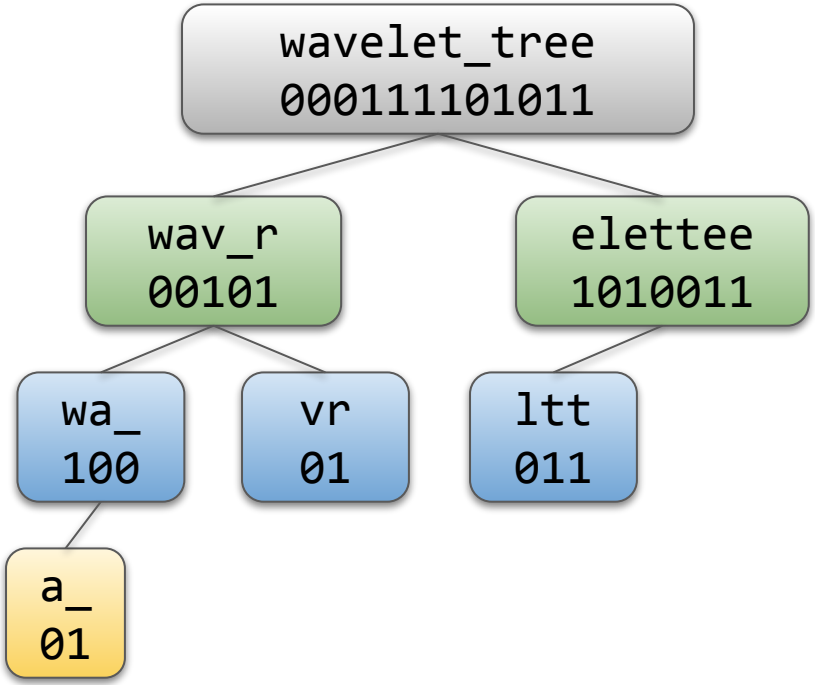
Huffman-Shaped Wavelet Trees



Char	Code
_	0100
a	0101
e	00--
l	0110
r	0111
t	10--
v	110-
w	111-

➔ **gaps** break (consecutive) levelwise representation

Huffman-Shaped Wavelet Trees



Char	Code
_	0001
a	0000
e	11--
l	100-
r	011-
t	101-
v	010-
w	001-

➔ inverting **canonical** Huffman codes causes gaps to move to the right

(levelwise representation remains consecutive)

Fast Levelwise Construction

for Huffman-Shaped WTs

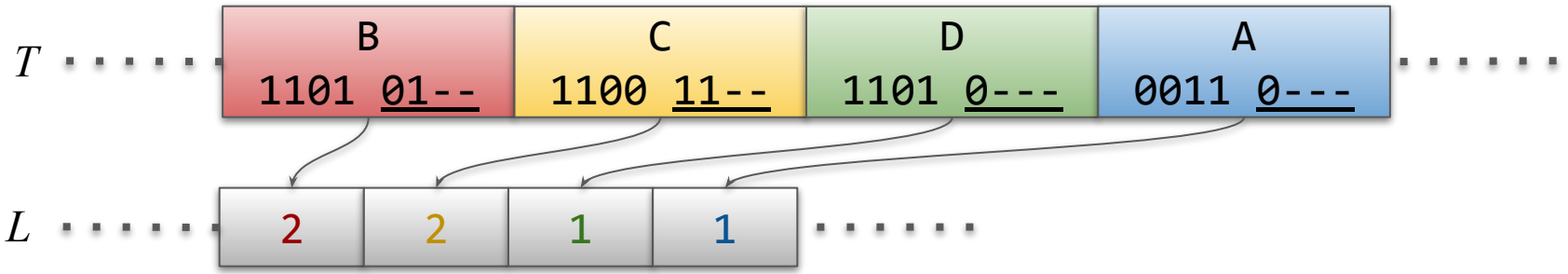
steps for each cluster:

#	Step	Running time
1	Cluster Extraction	$O(n)$
1a	Code Length Computation	$O(n)$
2	τ passes	$O(n\tau^2/\lg n) = O(n)$
2a	- Bit Extraction	- $O(n\tau/\lg n)$
2b	- List Filtering	- $O(n\tau/\lg n)$
2c	- List Splitting	- $O(n\tau/\lg n)$
3	Text Reshuffling	$O(n)$

$(\tau := \sqrt{\lg n})$

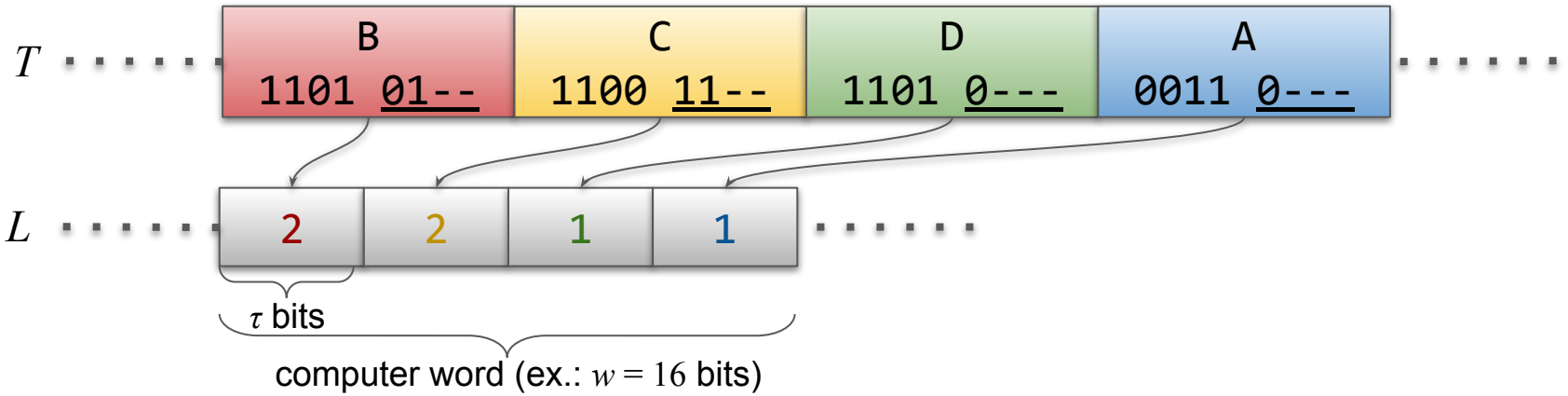
Code Length Computation

➔ store the **remaining** code length in the current cluster of each character to a **word-packed list** L



Code Length Computation

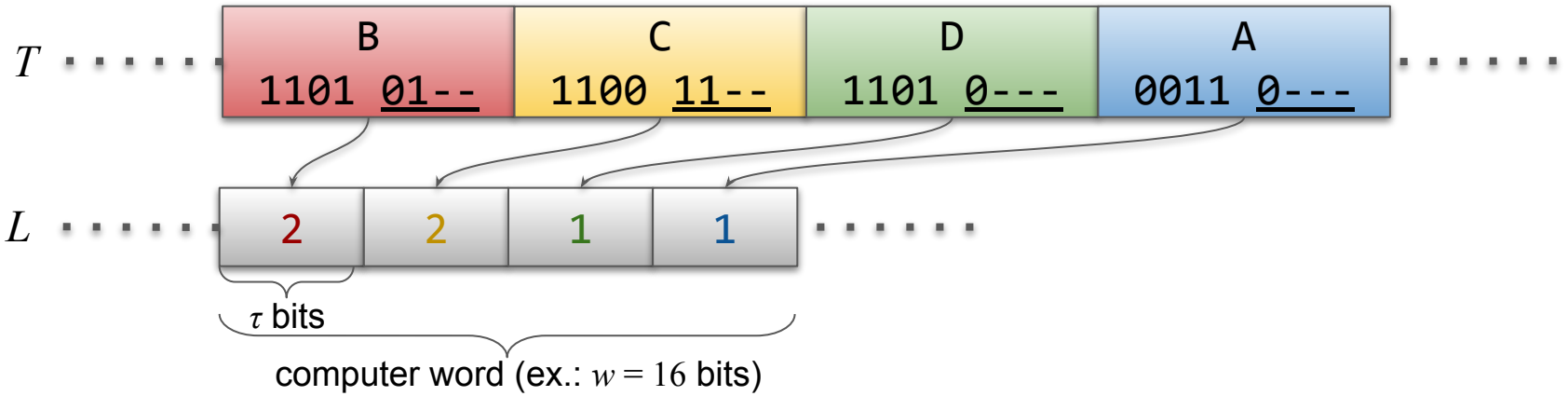
➔ store the **remaining** code length in the current cluster of each character to a **word-packed list** L



➔ we limit the lengths to τ so they fit into τ bits each

Code Length Computation

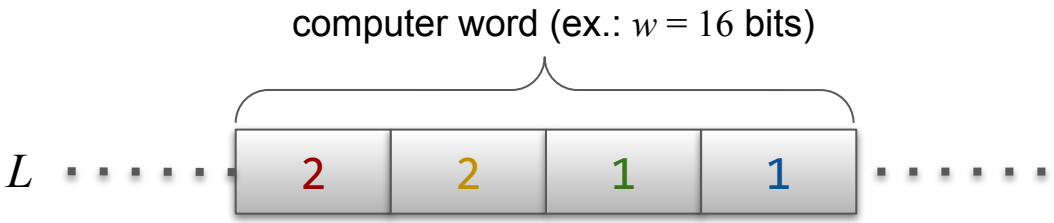
➔ store the **remaining** code length in the current cluster of each character to a **word-packed list** L



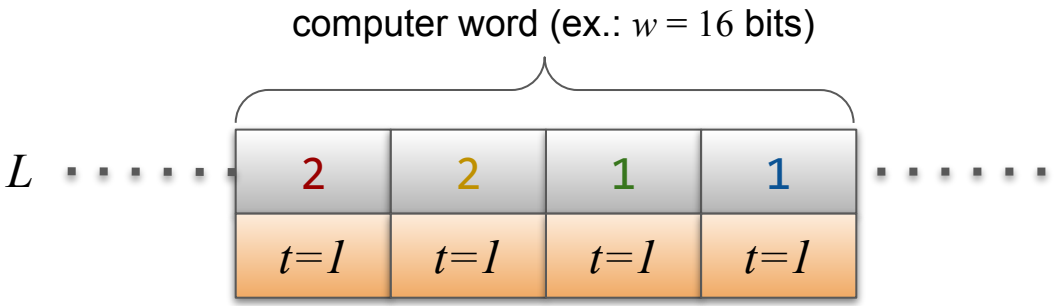
➔ we limit the lengths to τ so they fit into τ bits each

➔ left-to-right scan of T in time $O(n)$ ✓

List Filtering

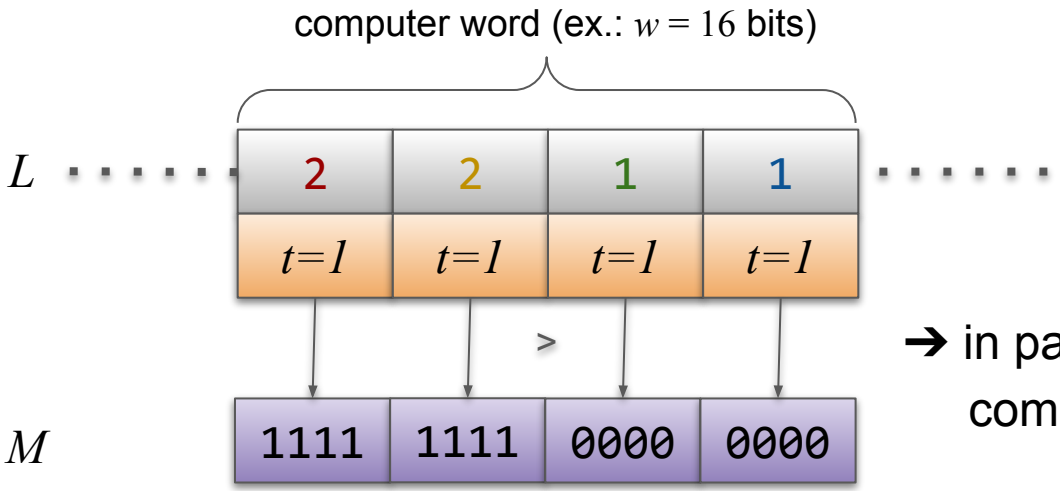


List Filtering



➔ in pass t, do a parallel greater-than comparison of lengths against t

List Filtering

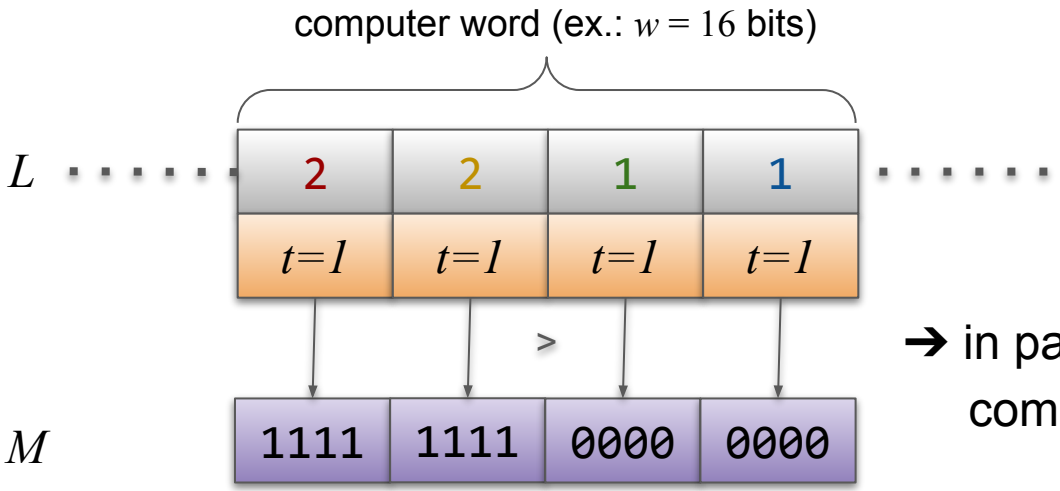


➔ in pass t, do a parallel greater-than comparison of lengths against t

➔ the result mask M is used to filter codes ending on level $\alpha\tau+t$ (`pext`)

➔ the parallel comparison can be done in constant time per word ✓

List Filtering



➔ in pass t, do a parallel greater-than comparison of lengths against t

➔ the result mask M is used to filter codes ending on level $\alpha\tau+t$ (`pext`)

➔ the parallel comparison can be done in constant time per word ✓

➔ if any code ends after pass t, then all following codes also end

(thanks to the inverted canonical Huffman codes)

Fast Levelwise Construction

for Huffman-Shaped WTs

steps for each cluster:

#	Step	Running time
1	Cluster Extraction	$O(n)$
1a	Code Length Computation	$O(n)$
2	τ passes	$O(n\tau^2/\lg n) = O(n)$
2a	- Bit Extraction	- $O(n\tau/\lg n)$
2b	- List Filtering	- $O(n\tau/\lg n)$
2c	- List Splitting	- $O(n\tau/\lg n)$
3	Text Reshuffling	$O(n)$

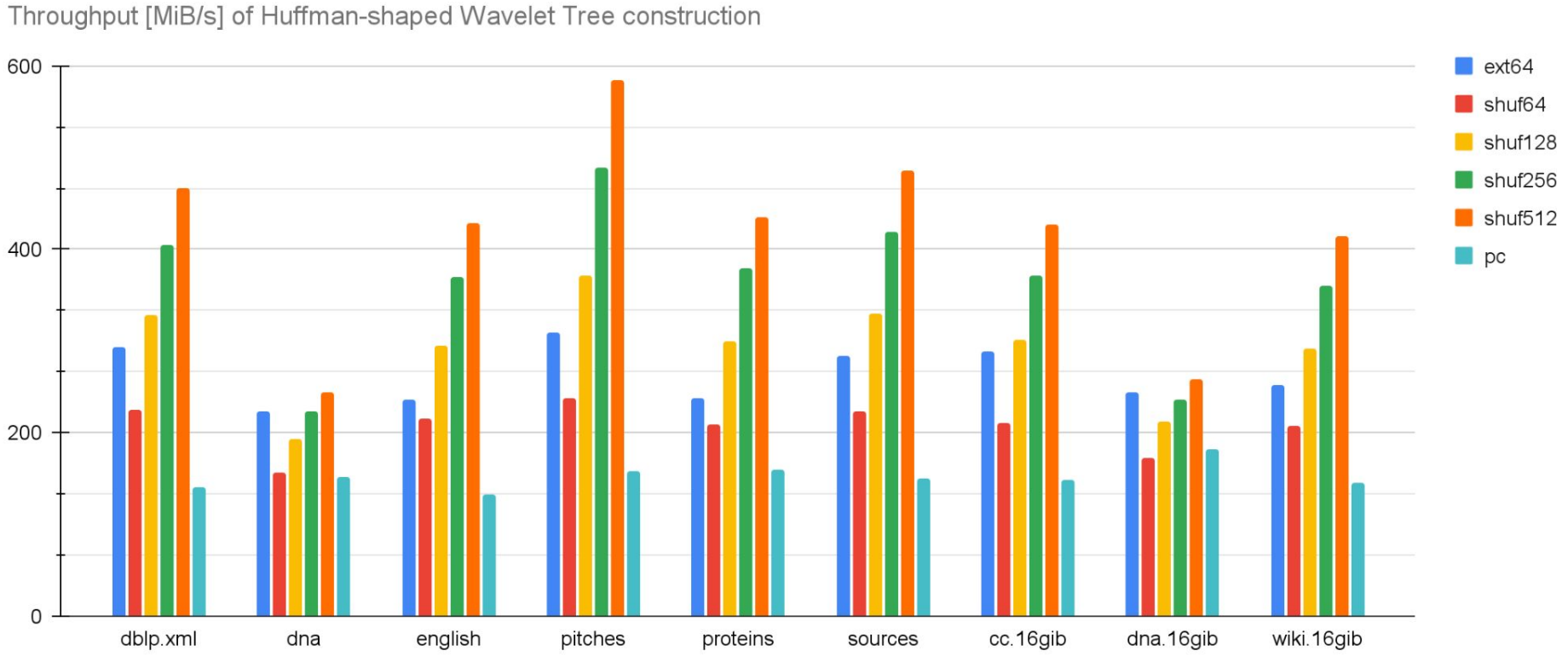
$(\tau := \sqrt{\lg n})$

Useful CPU Instructions

Name	Instruction	Brief	CPUID Flags
Population Count	`popcnt`	Count # of 1-bits in input word	`POPCNT`
Parallel Bit Extract	`pext`	Extract bits from word marked by mask; align in most significant bits	`BMI2`
Parallel Compare	`pcmp*` `vpcmp*`	Compare vector components; output bit vector containing results	`MMX` `AVX512*`
Compress	`vpcompress*`	Extract vector components (“pext for words”)	`AVX512*`
Bit Shuffle	`vpshuftbit*`	Gather bits from 64-bit subwords (“advanced pext”)	`AVX512_BITALG`
Permute	`pshufb` `vperm`	Permute vector components	`SSE3` `AVX512_BITALG`

➔ See Intel® Intrinsics Guide for details!

Experimental Results



CPU: Intel Core i9-11900KF with AVX-512, 3.5GHz (no turbo), 384 KiB L1

https://github.com/jptrn/mawt