
CS 267

Lecture 26: Computational Biology

Aydin Buluc

Slide credits: Evangelos Georganas, Carl Kingsford, Giulia Guidi, and Christian Nørgaard Storm Pedersen

<https://sites.google.com/lbl.gov/cs267-spr2019/>

Outline

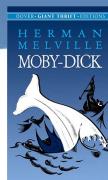
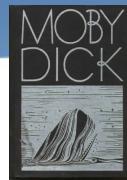
Computational Biology is a huge field; some will claim it is not even a coherent field. It has both simulation (e.g. cell simulations) and data analysis (e.g. –omics) aspects.

This lecture focuses on genomics aspects:

- **Genome (and metagenome) assembly problems**
- Indexing with hash tables
- Pairwise sequence alignment
- Indexing with suffix arrays

De novo Genome Assembly

1. Three copies of the same novel.



1. Three copies of the same DNA.



2. Some text from the novel. All pages will be randomly cut into strips of characters. Random **typos (errors)** throughout each novel.

For all men tragically great are made so through a certain morbidness... all mortal greatness is but disease.

2. Some part of the DNA sequence. It will be read into strips. There are random **errors** throughout the sequence.

ACCGTAGCAAAACCGGGTAGTCATACTACTACGTACTCATCT

3. A few strips of characters from one page.

For a ally great
great are made so all men tragically g

3. The sequence is read into smaller pieces (**reads**). Can not read whole DNA sequence *in one go*.

ACCGTAGCAA AAACCGGGTA TAGTCATACT
AAACCGGGTA ACTACGTACT

4. All of the strips of characters from the 3 novels.



4. All reads

[Sequence of short black and blue horizontal bars representing individual DNA reads]

5. Every strip must be assembled as shown here to create a single copy of the novel.

For all men tragically great are made so
For a ally great
great are made so
all men tragically g

5. Reconstruct original DNA sequence from the read set.

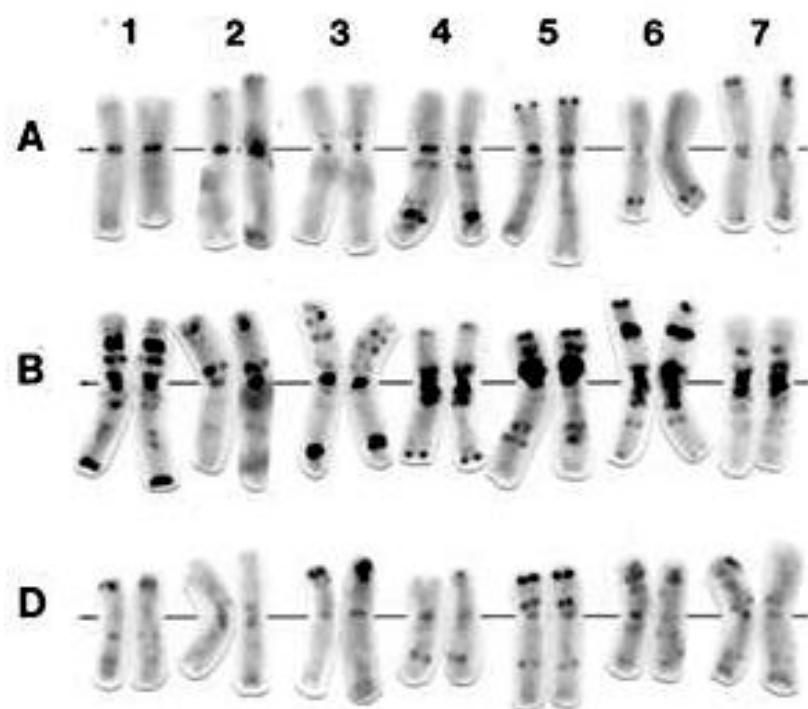
ACCGTAGCAAAACCGGGTAGTCATACTACTACGTACTCATCT
ACCGTAGCAA GTAGTCATACT
AAACCGGGTA CTACTACGTAC
CGTACTCATCT

De novo Genome Assembly is hard

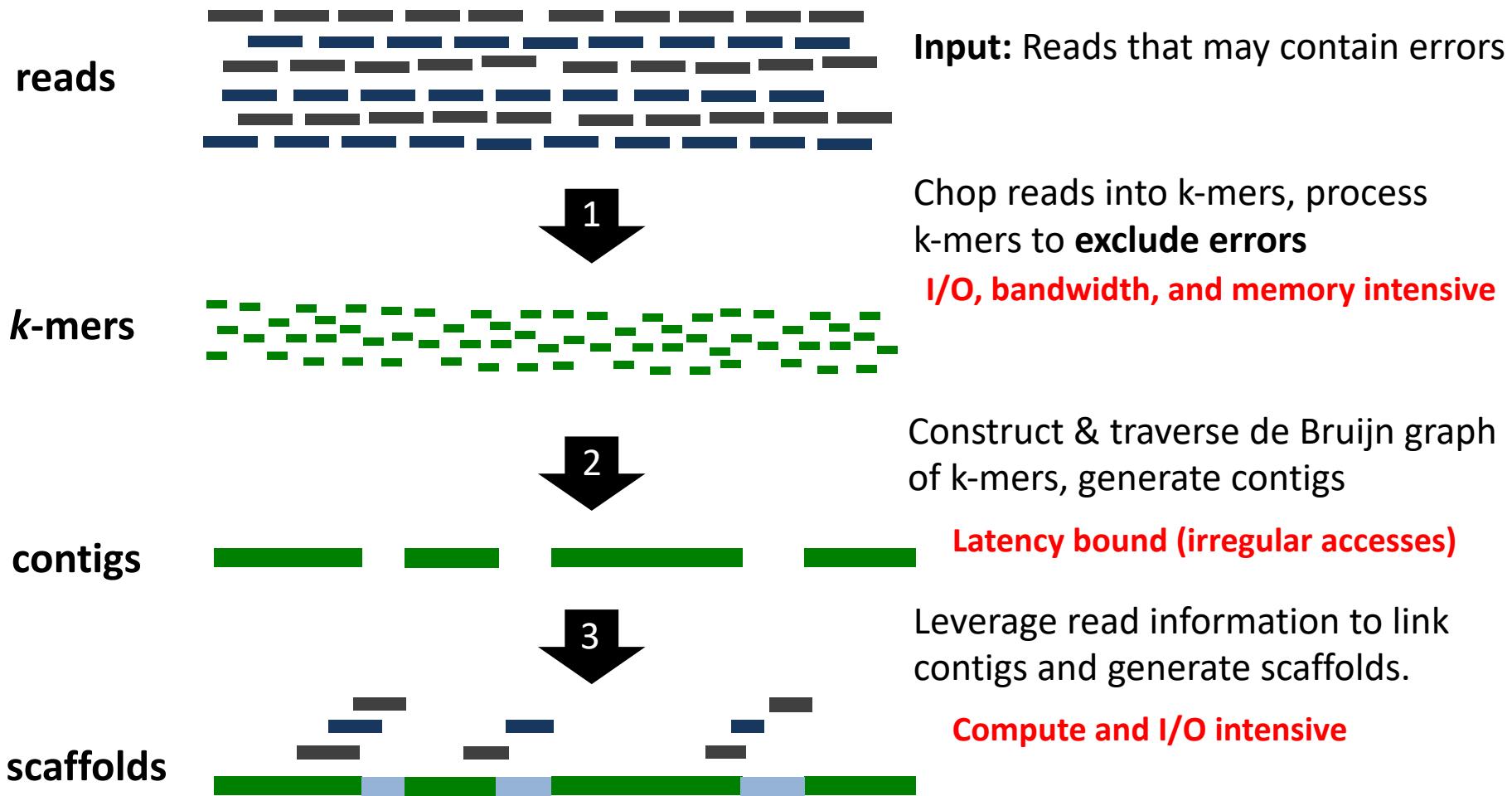
- There is no genome reference!
 - In principle we want to reconstruct unknown genome sequence.
- Reads are significantly shorter than whole genome.
 - Reads consist of 20 to 30K bases
 - Genomes vary in length and complexity – up to 30G bases
- Reads include **errors**.
- Genomes have repetitive regions.
 - Repetitive regions increase genome complexity.

Genomes vary in size

- Switchgrass: 1.4 Giga-base pairs (Gbp)
- Maize: 2.4 Gbp
- Miscanthus: 2.5 Gbp
- **Human genome: 3 Gbp**
- Barley genome: 7 Gbp
- **Wheat genome: 17 Gbp**
- Pine genome: 20 Gbp
- Salamander: 20-30 Gbp

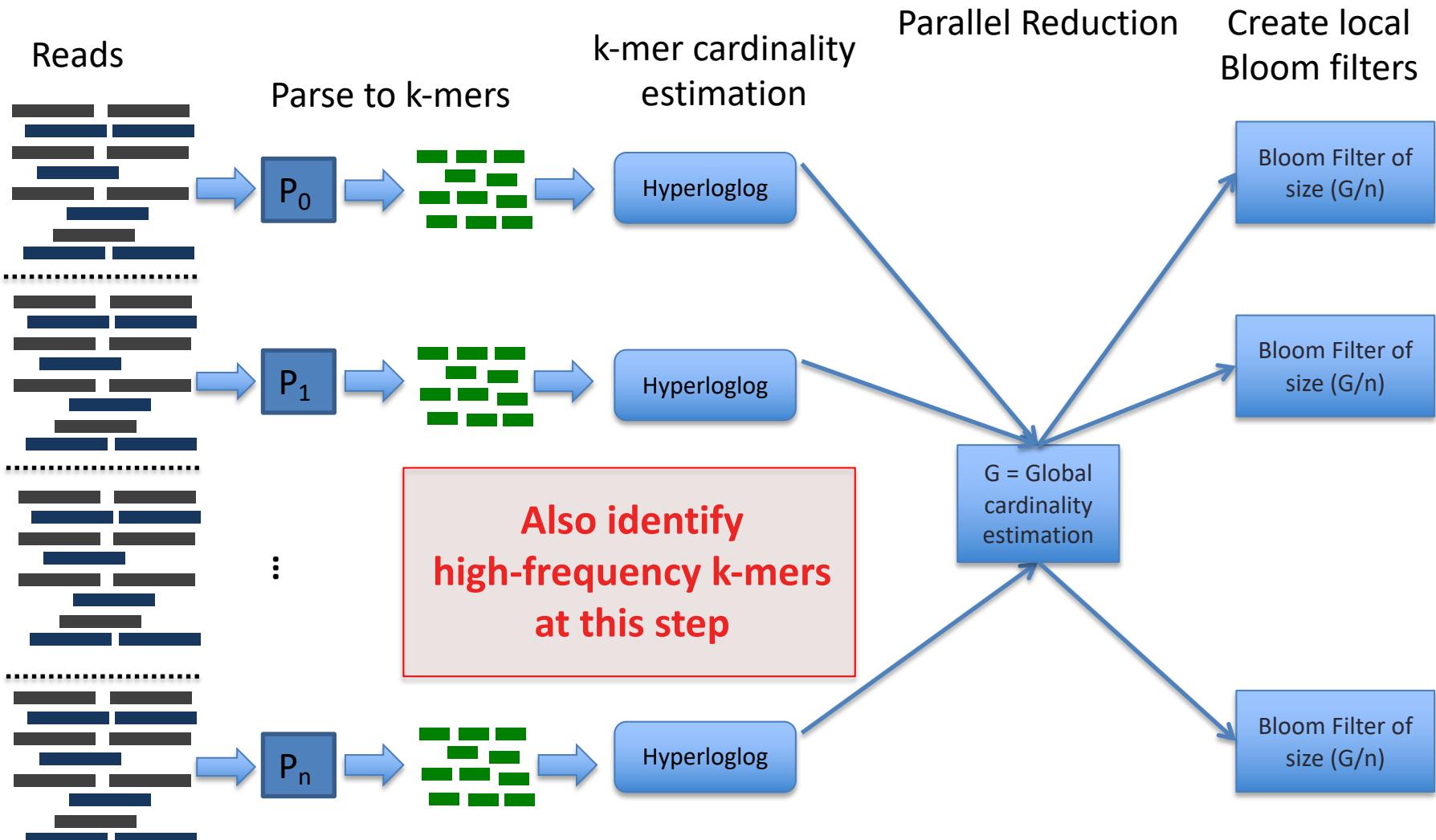


Genome Assembly a la HipMer

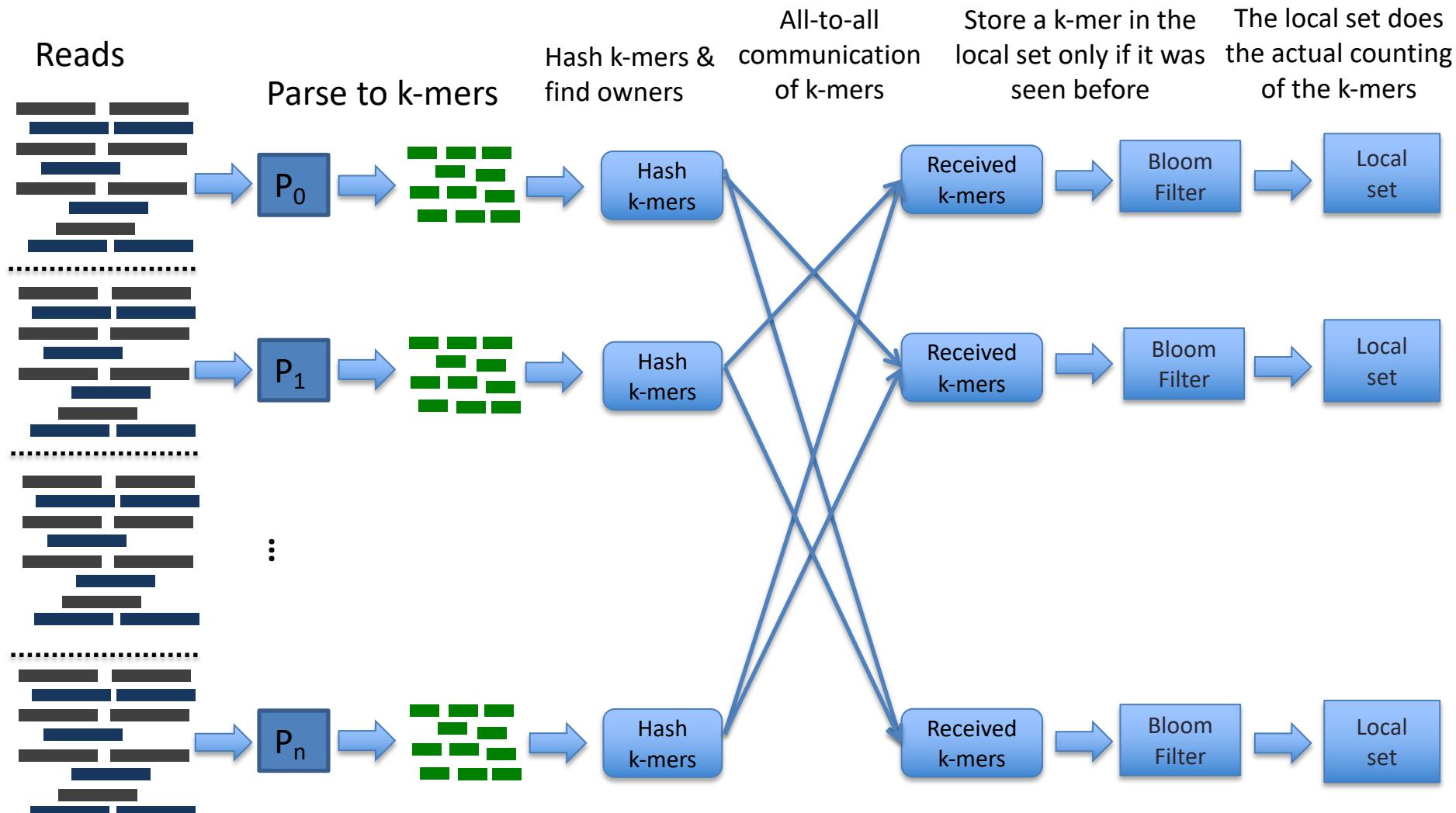


Georganas, E., Buluç, A., Chapman, J., Hofmeyr, S., Aluru, C., Egan, R., Oliker, L., Rokhsar, D. and Yellick, K., HipMer: an extreme-scale de novo genome assembler. In SC'15.

Parallel k-mer analysis: pass 1



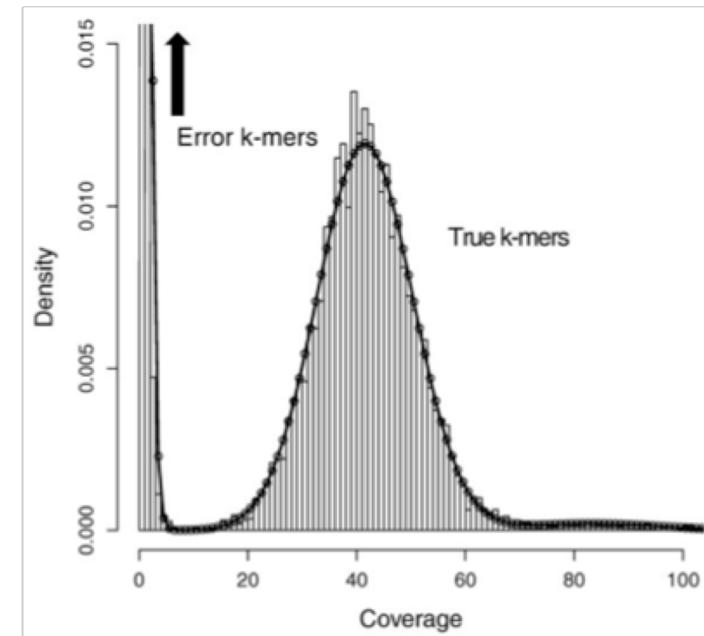
Parallel k-mer analysis: pass 2



Why use a Bloom filter?

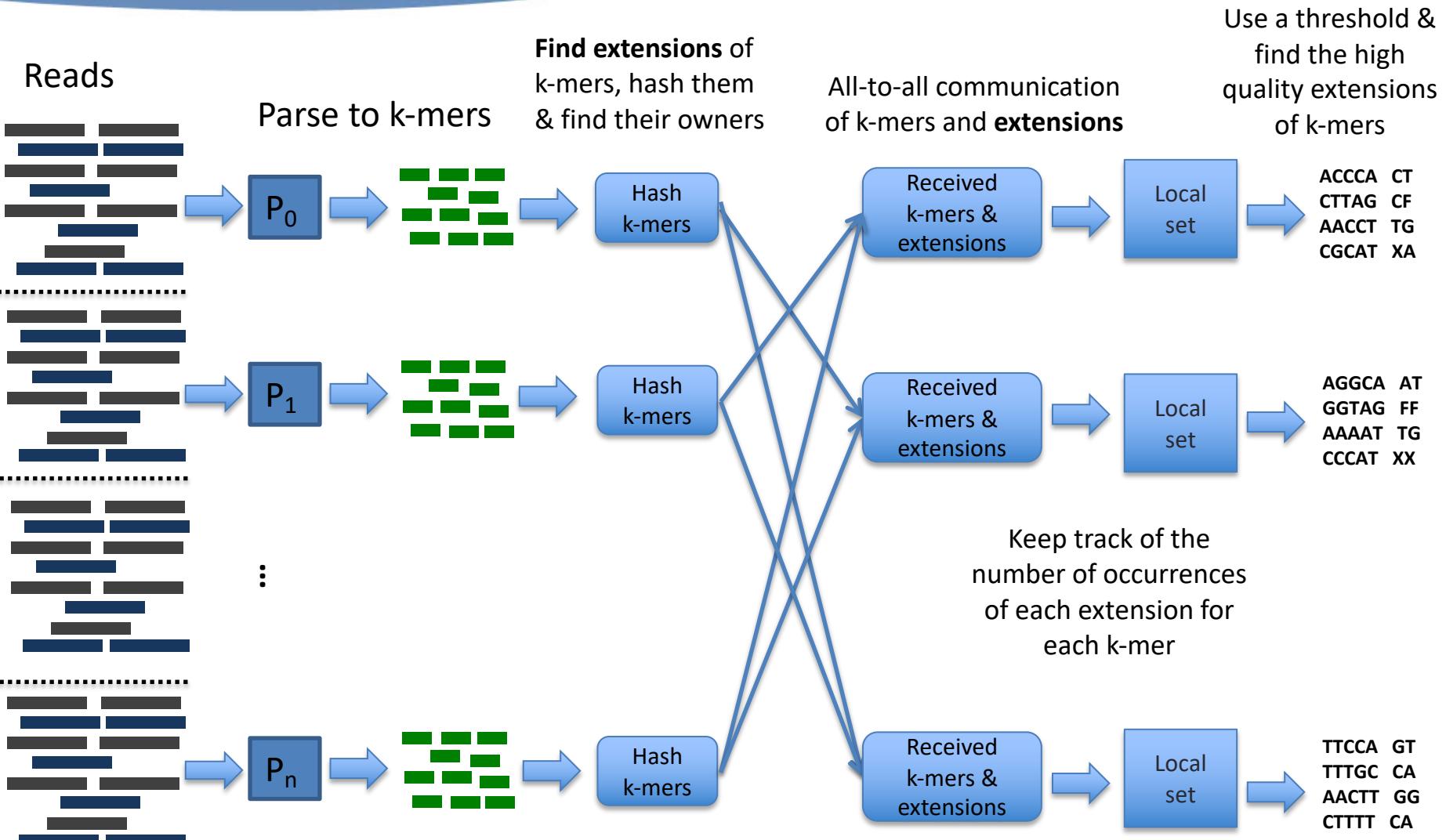
Bloom filter is a *probabilistic* data structure used for membership queries

- Given a bloom filter, we can ask:
“Have we seen this k-mer before?”
- No false negatives.**
- May have false positives
(in practice 5% false positive rate)



k-mers with frequency =1 are useless (either error or can not be distinguished from error), and can safely be eliminated.

Parallel k-mer analysis: pass 3

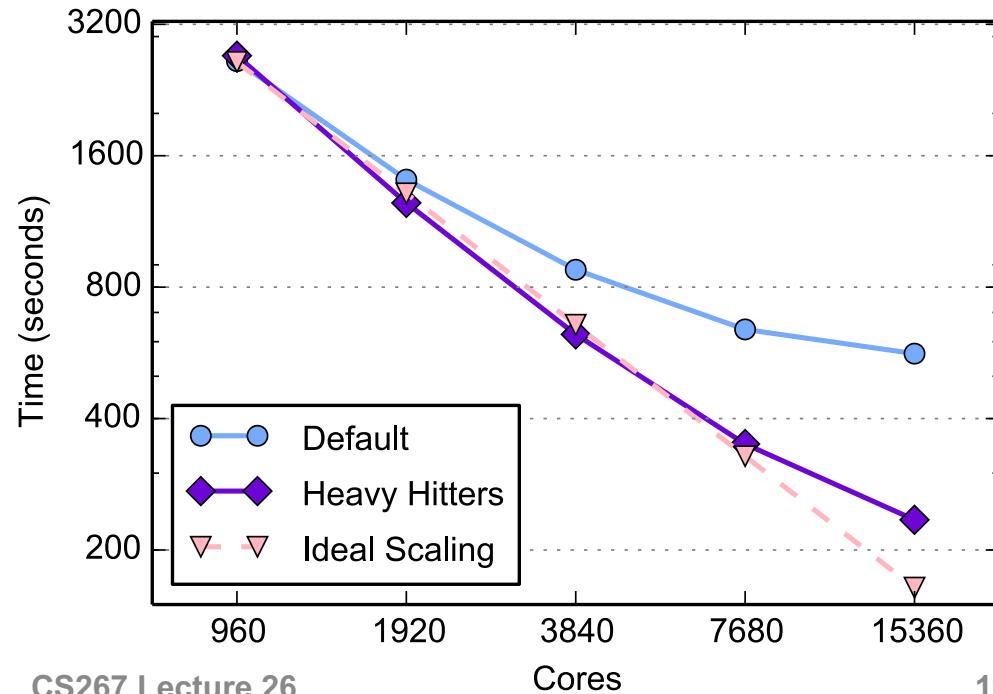


High-frequency k-mers

Long-tailed distribution for genomes with repetitive content:

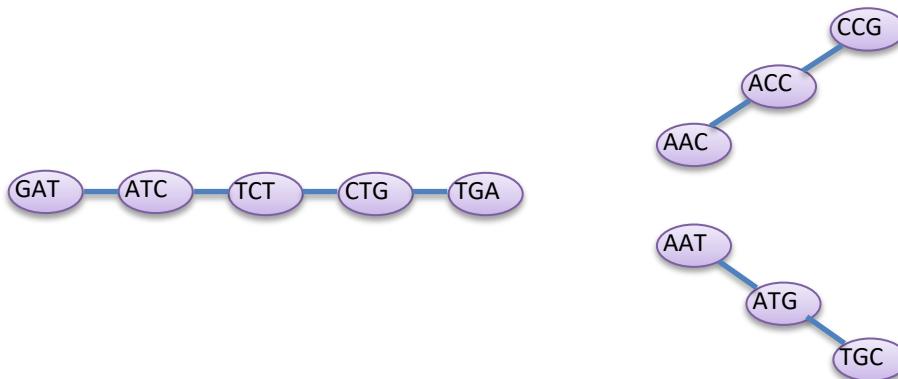
- The maximum count for any k-mer in the wheat dataset is **451 million**
- Our original scheme (SC'14) was “owner counts”, after an all-to-all
- Counting an item w/ 451 million occurrences alone is **load imbalanced**

Solution: Quickly identify high-frequency k-mers using minimal communication during the “cardinality estimation” step and treat them specially by using local counters.



Parallel De Bruijn Graph Construction

- In Meraculous, the de Bruijn graph is represented as a hash table
- K-mers are both *nodes in the graph & keys in the hash table*
- An edge in the graph connects two nodes that overlap in k-1 bases
- The edges in the graph are put in the hash table by storing the extensions of the k-mers as their corresponding values



Parallel De Bruijn Graph Construction

Graph construction, traversal, and all later stages are written in UPC to take advantage of its global address space

Input: k-mers and their high quality extensions

Read k-mers & extensions

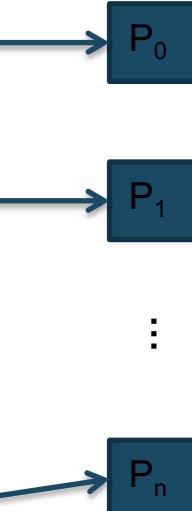
Store k-mers & extensions

Distributed Hash table

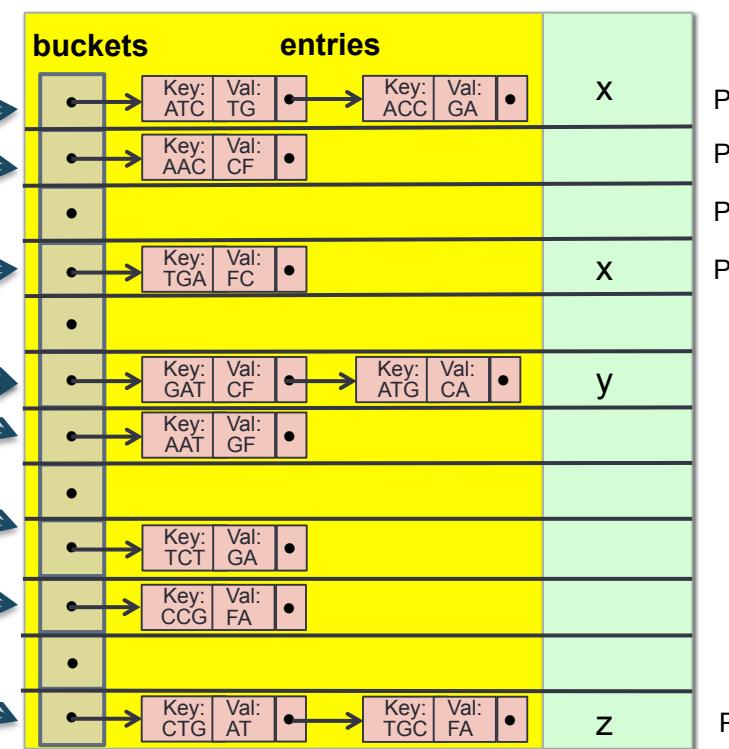
Shared

Private

AAC	CF
ATC	TG
ACC	GA
...	
TGA	FC
GAT	CF
AAT	GF
...	
ATG	CA
TCT	GA
...	
CCG	FA
CTG	AT
TGC	FA



Fine-grained communication & fine-grained locking required

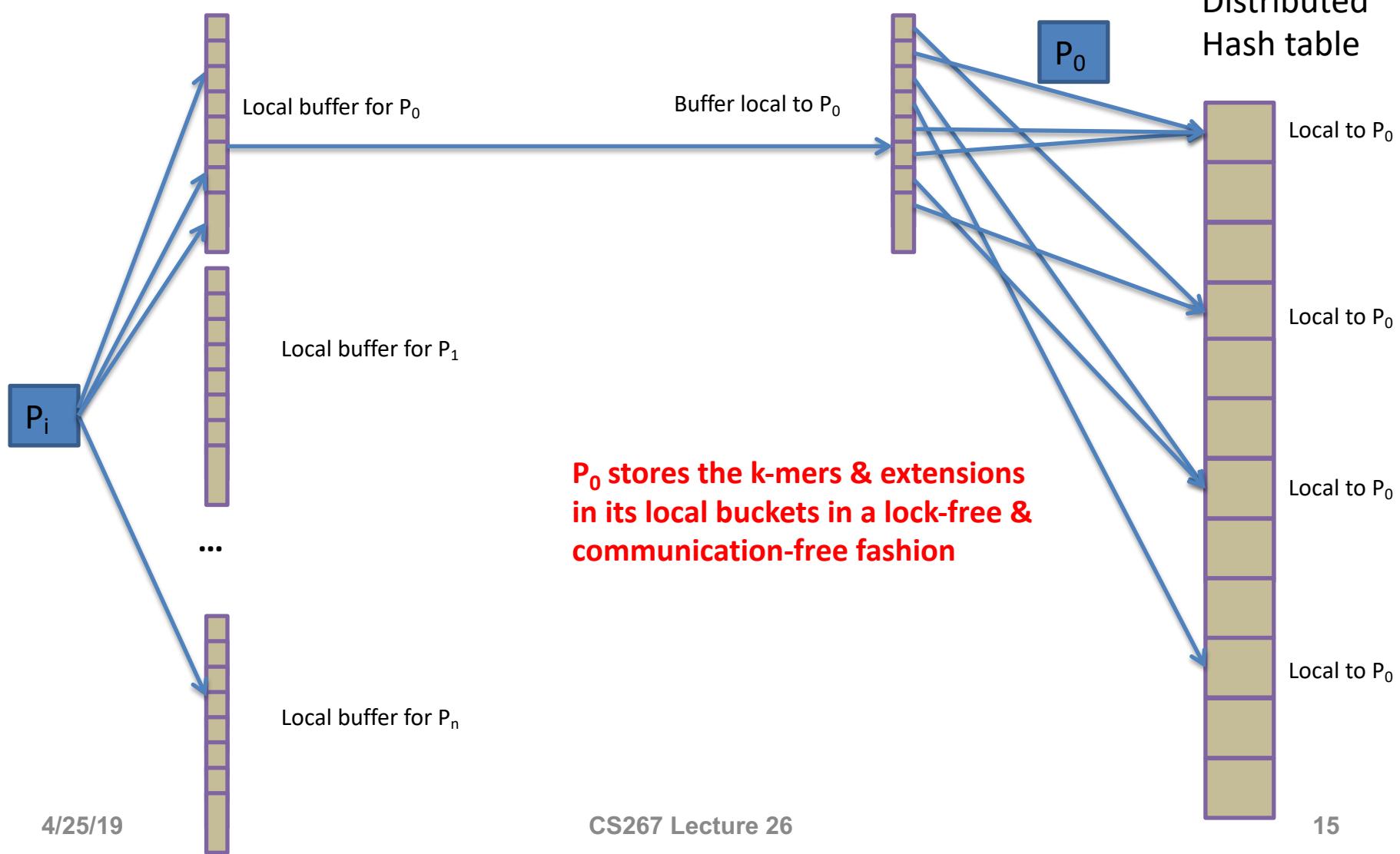


Parallel Graph Construction Challenge

- **Challenge 1: Hash table for de Bruijn graph is huge (at least multiple terabytes)**
 - **Solution:** Distribute the graph over multiple processors.
- **Challenge 2: Parallel hash table construction introduces communication and synchronization costs**
 - **Solution:** Split the construction in two phases and aggregate messages → **10x-20x performance improvement.**

Georganas, E., Buluç, A., Chapman, J., Oliker, L., Rokhsar, D. and Yelick, K. Parallel de bruijn graph construction and traversal for de novo genome assembly. In SC'14.

Aggregating stores optimization

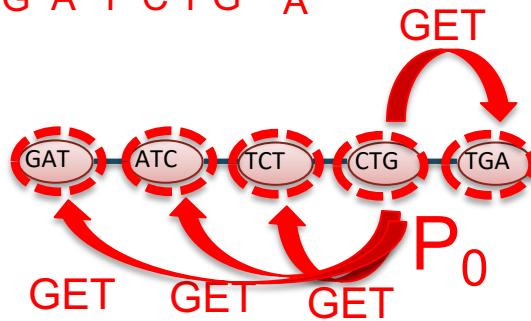


Parallel De Bruijn Graph Traversal

Contig:

G A T C T G A

GET

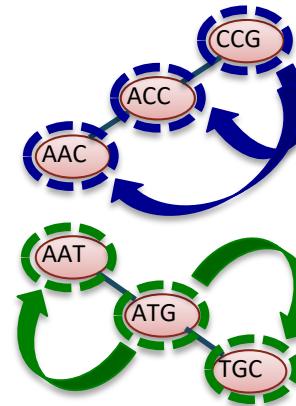


Algorithm: Pick a random k-mer and expand connected component by consecutive lookups in the distributed hash table.

Contig:

A A C C G

P_1



Contig:

A A T G C

P_2

Private

Shared

buckets

entries

X
 P_0
 P_1
 P_2
 P_3
...
 P_n

Global Address Space

	Shared	Private
P_0	Key: ATC Val: TG Key: AAC Val: CF	X
P_1	Key: TGA Val: FC	X
P_2	Key: GAT Val: CF Key: AAT Val: GF	y
P_3	Key: ATG Val: CA	
...		
P_n	Key: TCT Val: GA Key: CCG Val: FA	z
	Key: CTG Val: AT Key: TGC Val: FA	

Parallel De Bruijn Graph Traversal

Goal:

- Traverse the de Bruijn graph and find UU contigs (chains of UU nodes), *or alternatively*
- find the connected components which consist of the UU contigs.
- **Main idea:**
 - Pick a seed
 - Iteratively extend it by consecutive lookups in the distributed hash table

Parallel De Bruijn Graph Traversal

Assume *one* of the UU contigs to be assembled is:

CGTATTGCCAATGCAACGTATCATGGCCAATCCGAT

Parallel De Bruijn Graph Traversal

Processor P_i picks a random k-mer from the distributed hash table as seed:

CGTATTGCCAATG**CAACGTATC**A**T**GGCCAATCCGAT

P_i knows that forward extension is A

P_i uses the last $k-1$ bases and the forward extension and forms: CAACGTATCA

P_i does a lookup in the **distributed hash table** for CAACGTATCA

P_i iterates this process until it reaches the “right” endpoint of the UU contig

P_i also iterates this process backwards until it reaches the “left” endpoint of the UU contig

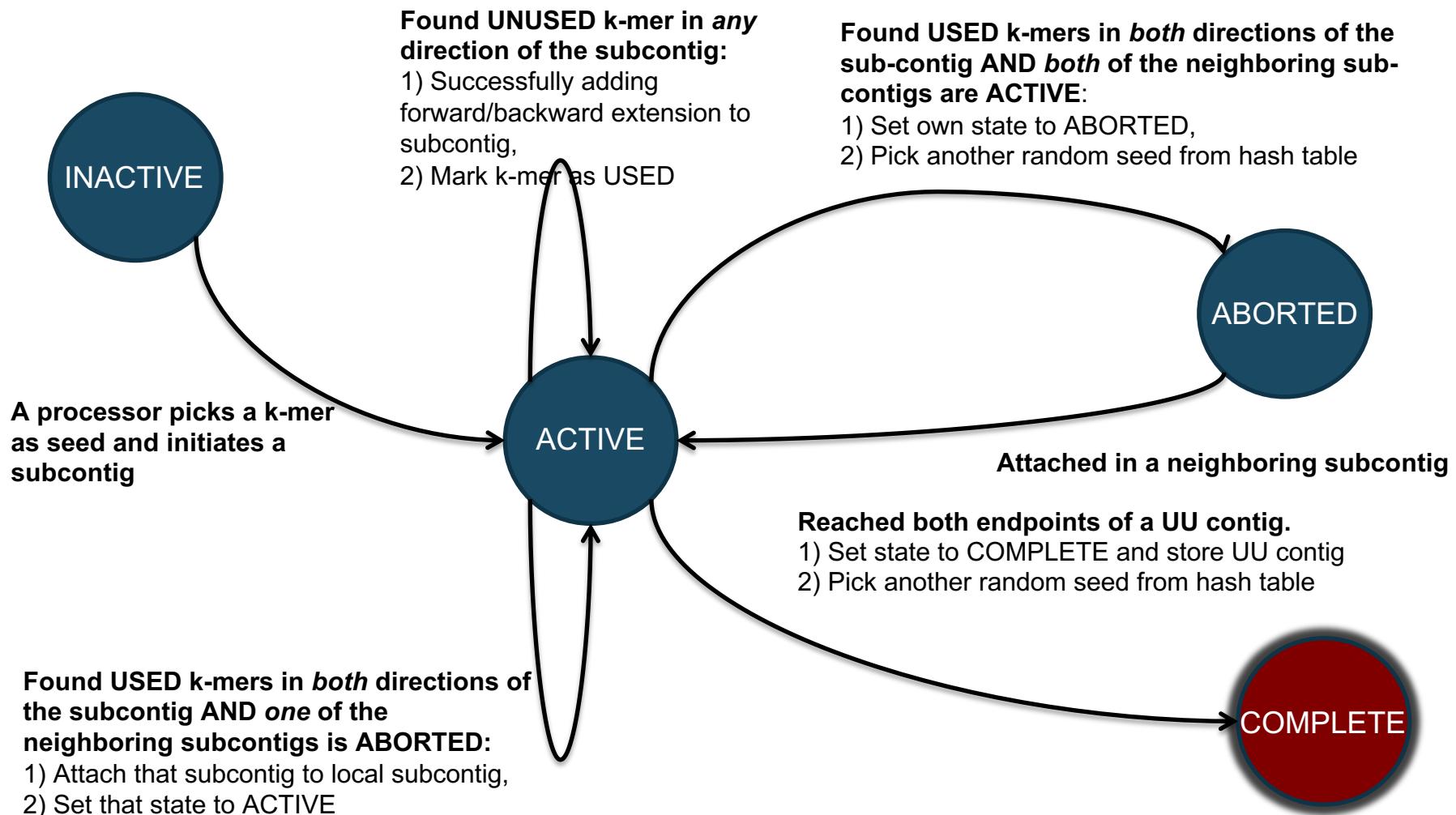
Multiple processors on the same UU contig



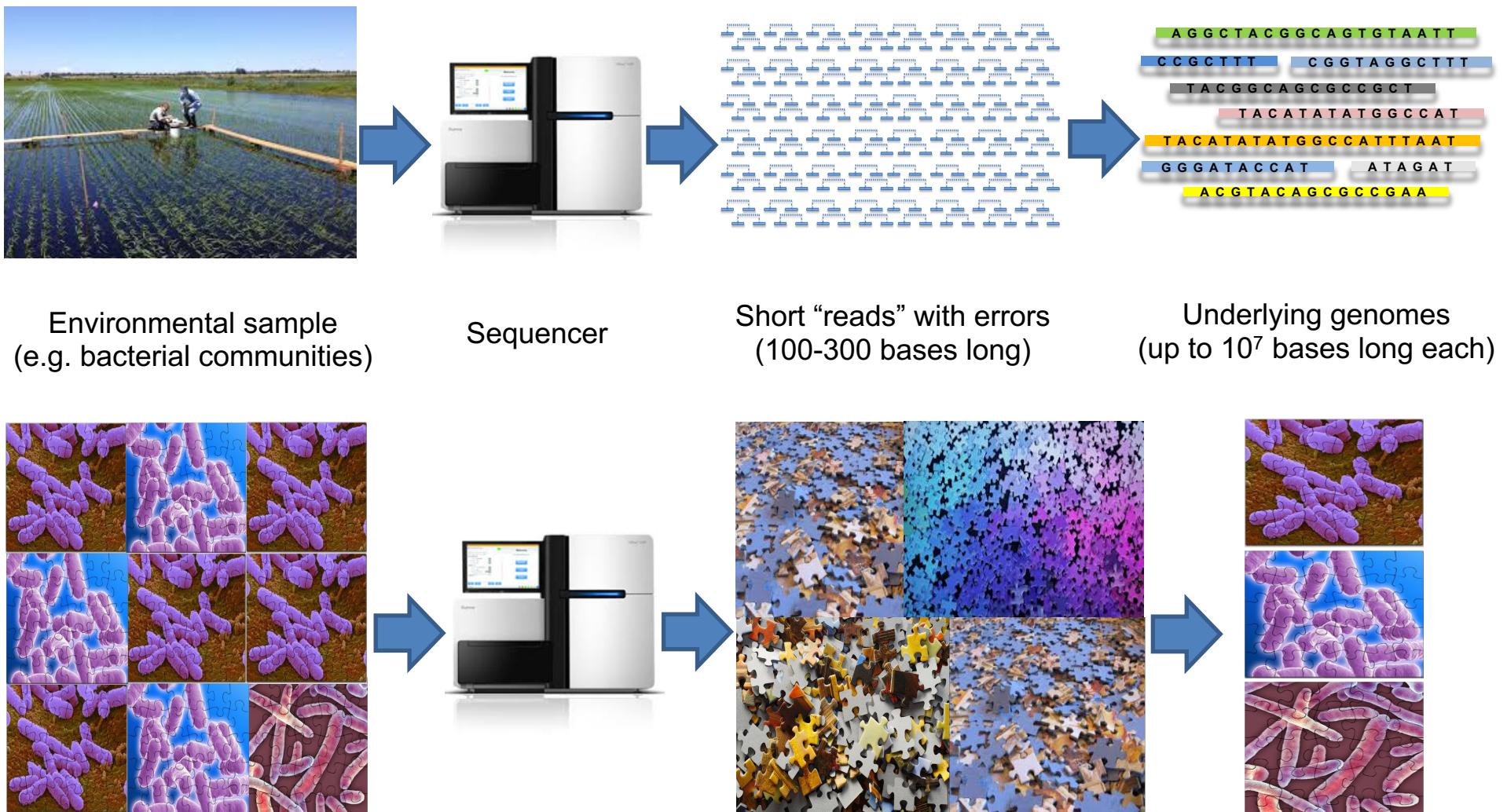
However, processors P_i , P_j and P_t might have picked initial seeds from the same UU contig

- Processors P_i , P_j and P_t have to collaborate and concatenate subcontigs in order to avoid redundant work.
- **Solution:** lightweight synchronization scheme based on a state machine

Lightweight Synchronization Protocol

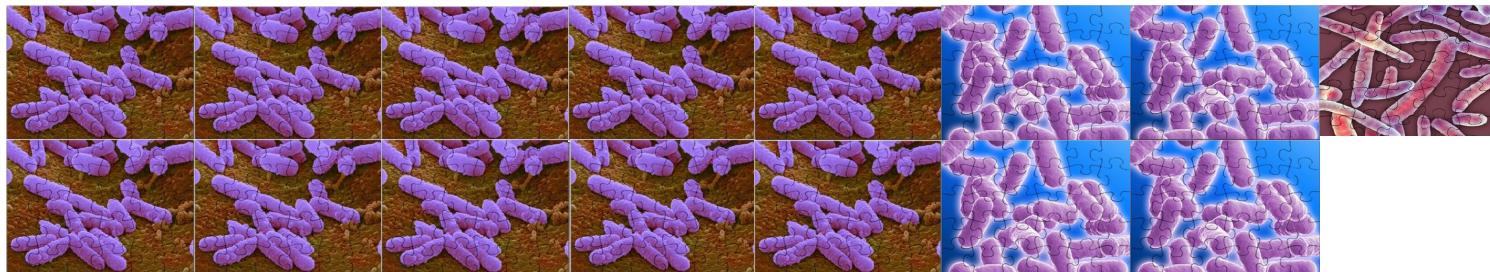


The metagenome assembly problem

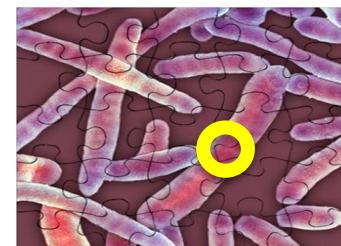
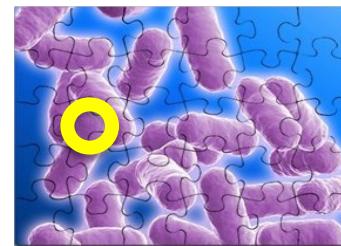
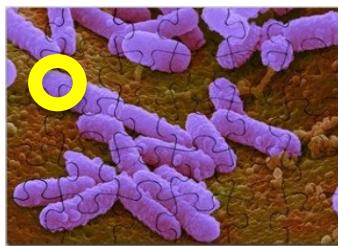


De novo metagenome assembly is harder !!!

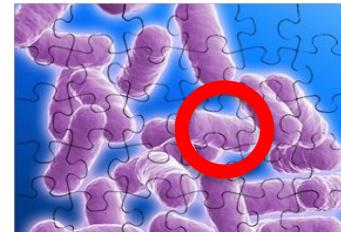
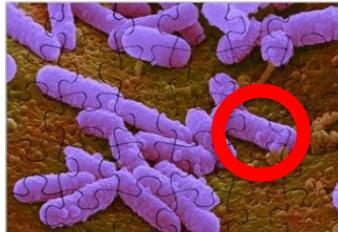
- Variable frequency (abundance) of the genomes within the sample



- Repeated sequences across genomes



- Polymorphism within species (“similar but not identical”)



Jigsaw puzzle analogy



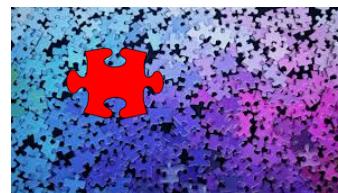
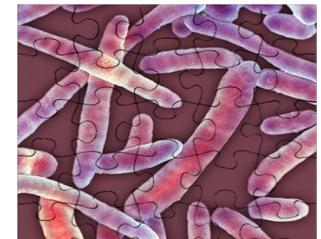
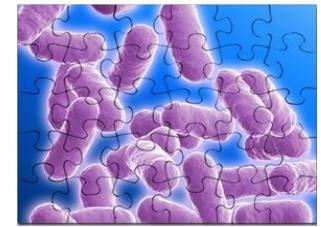
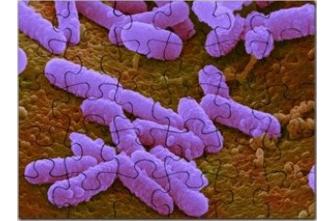
X 10



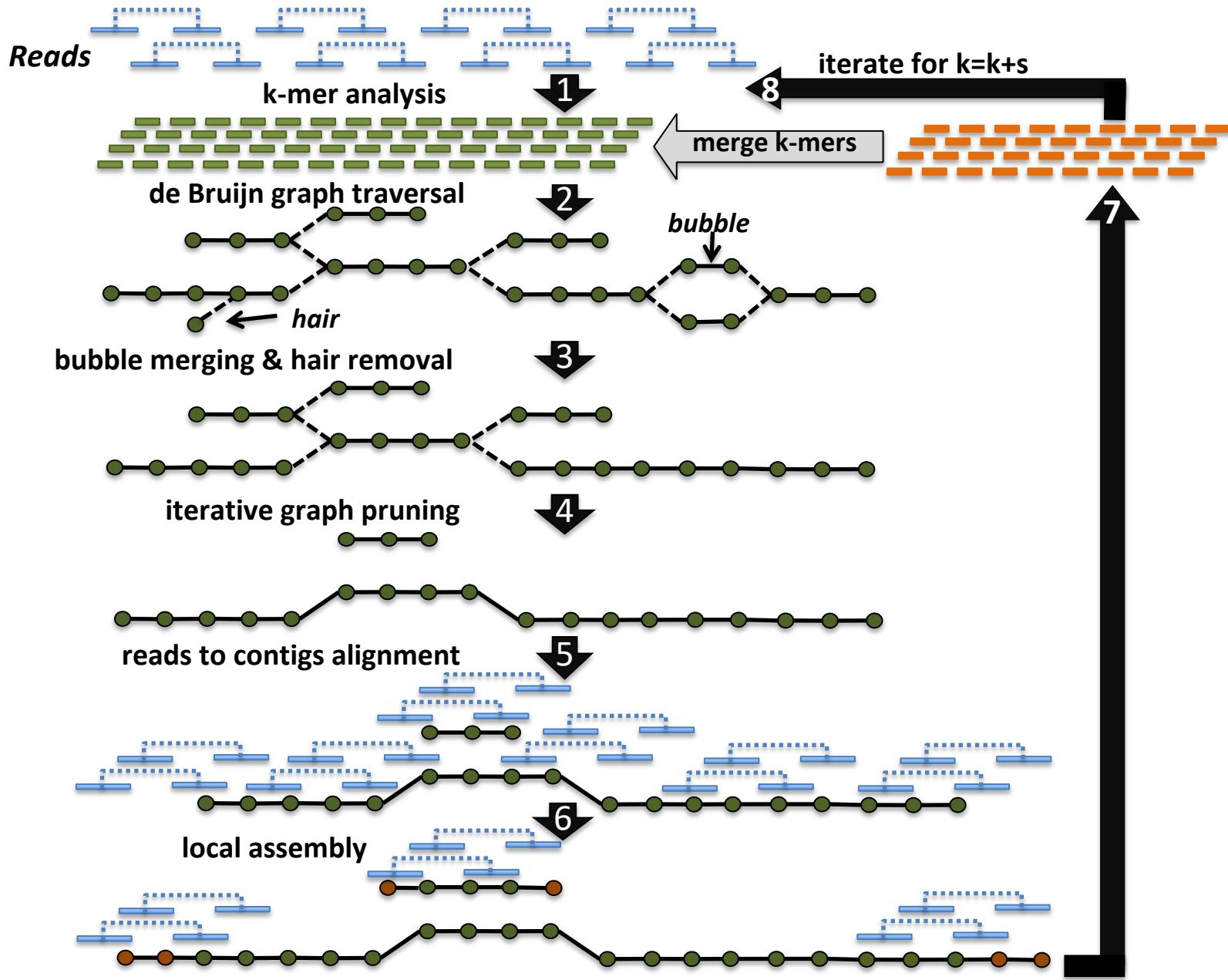
X 5



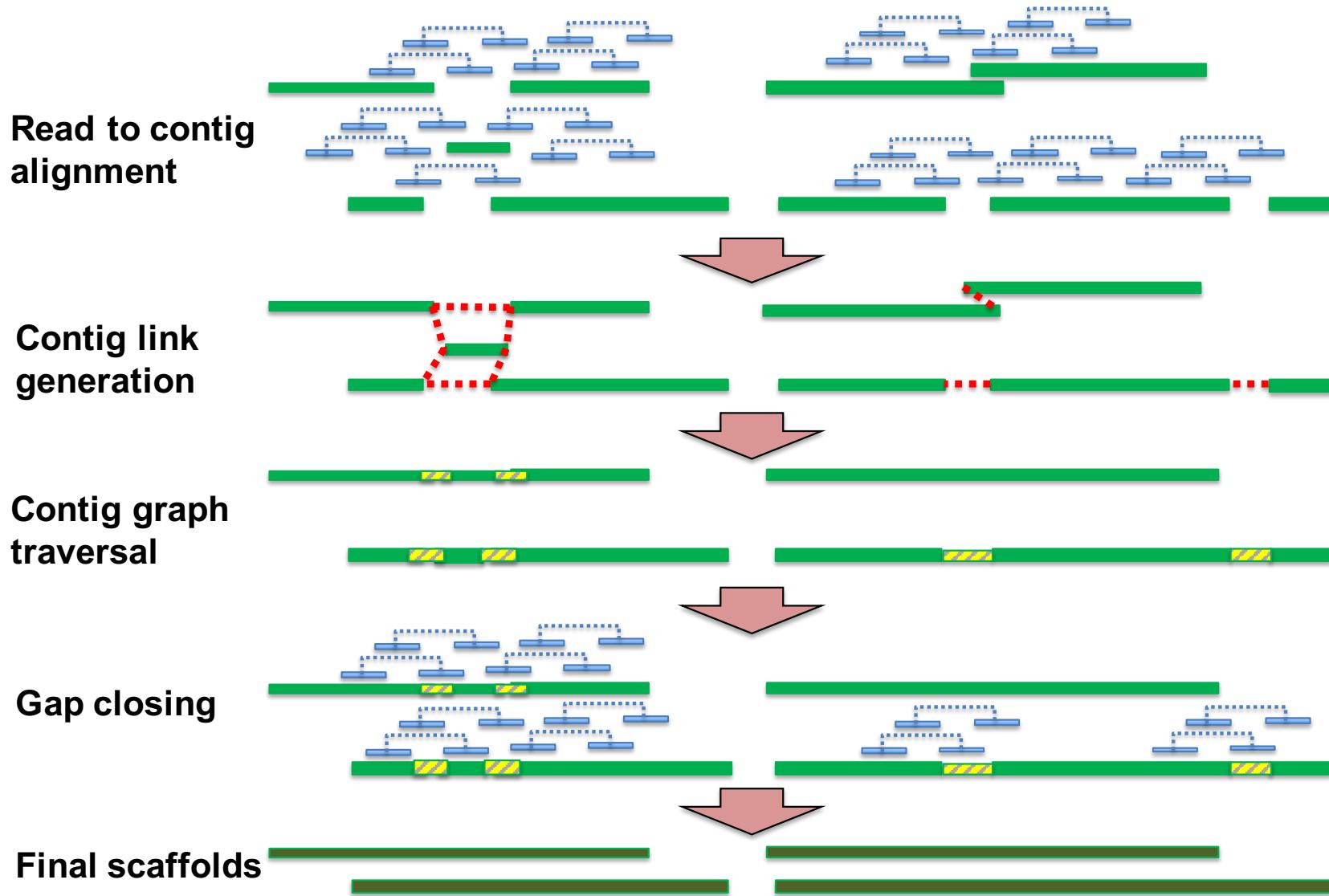
X 1



Iterative contig generation algorithm



The MetaHipMer scaffolding algorithm



Parallelization strategy

- The involved algorithms are inherently irregular
 - irregular & all-to-all communication patterns

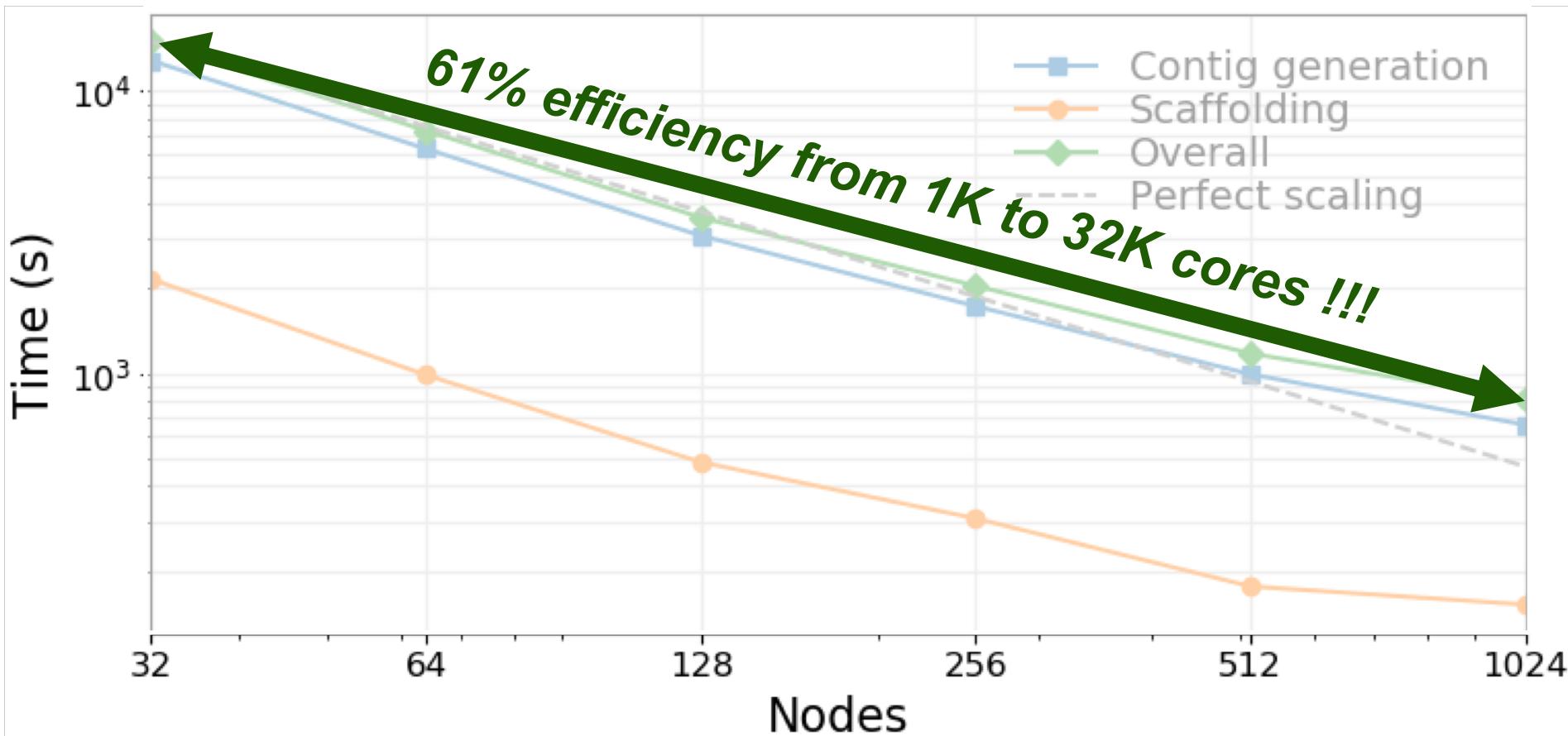


- **Core ideas for efficient parallelization**

1. Design parallel algorithms using a PGAS paradigm
2. Use distributed hash tables
3. Optimize common use-cases of hash tables
4. Understand data/bottlenecks and iterate 1, 2, 3

PGAS: Partitioned Global Address Space

MetaHipMer strong scaling



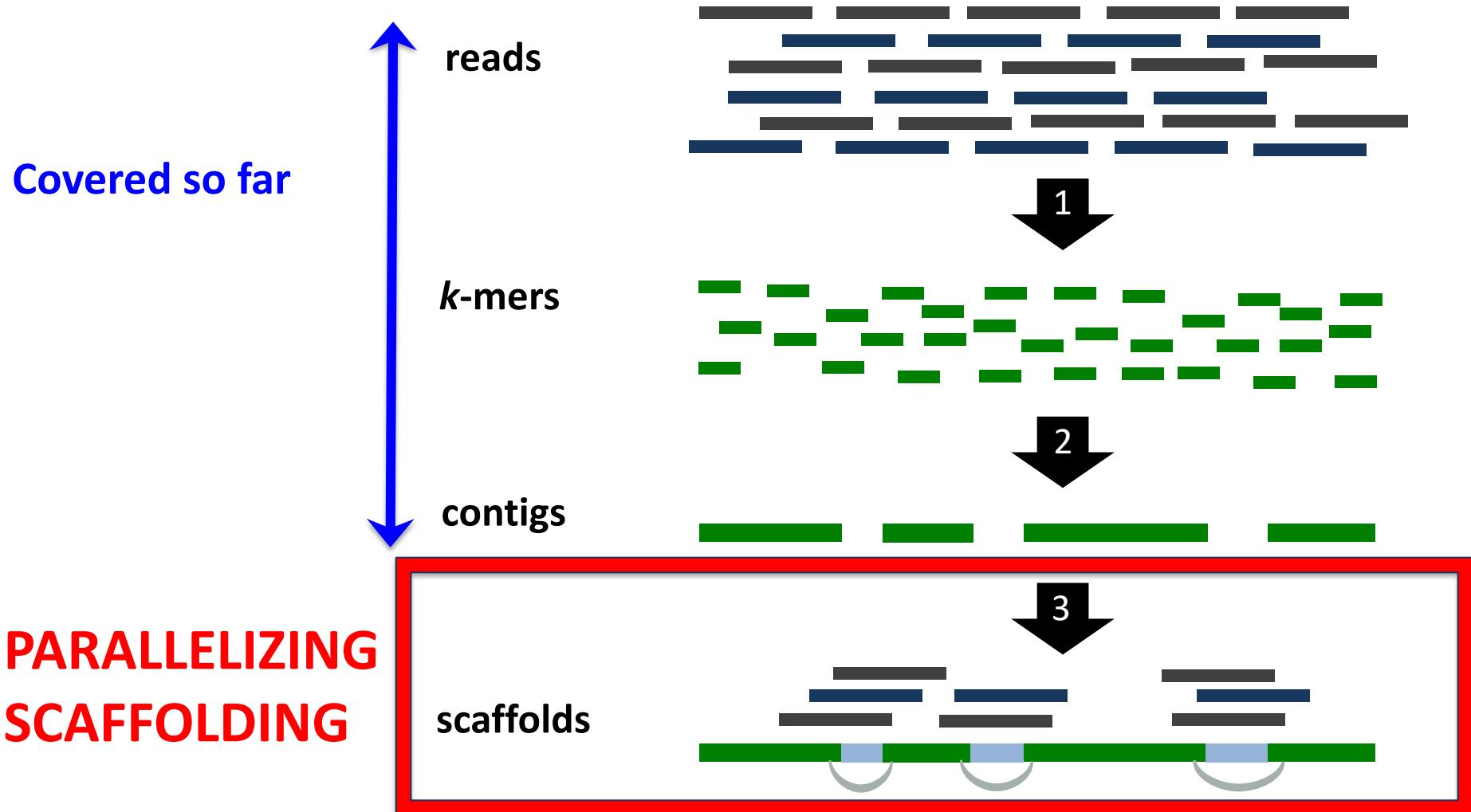
- Run on a subset of the Wetlands (soil metagenome) dataset
 - Real, high-complexity dataset
- Experimental platform: a XC40 Cray system (Cori @ NERSC)
- **Strong scaling efficiency 61% from 1K to 32K cores !!!**

Grand challenge problem: *Assembly of the full Wetlands dataset*

- Full Wetlands dataset:
 - Massive-scale dataset, consisting of 2.6 Tbytes of raw reads.
- Assembling such datasets with state-of-the-art tools is **intractable**
 - shared memory tools, hence limited available memory
 - previous approaches assembled **subsampled datasets**
 - **compromised quality**
- **MetaHipMer on 512 nodes assembled FULL dataset in 3h25mins**
- To the best of our knowledge this is the **largest, high-quality de novo metagenome assembly completed to date.**

E. Georganas, R. Egan, S. Hofmeyr, E. Goltsman, B. Arndt, A. Tritt, A. Buluç, L. Oliker, K. Yelick.
Extreme Scale De Novo Metagenome Assembly. **Best Paper Nominee (2 out of 288).** In
Proceedings of ACM/IEEE Supercomputing Conference (SC'18), 2018.

Alignment for De novo Genome Assembly

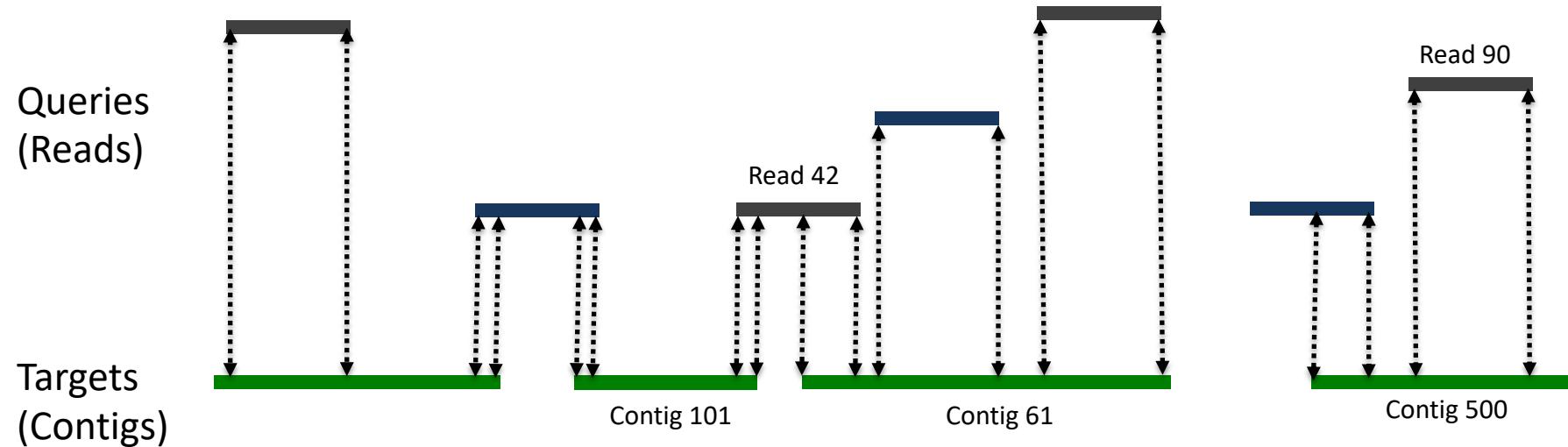


Outline

- Genome (and metagenome) assembly problems
- **Indexing with hash tables**
- Pairwise sequence alignment
- Indexing with suffix arrays

Aligning queries to target sequences

- A query and a target should match in at least **k** bases in order to be aligned
- We call **seed** a substring of a sequence (query or target) with length equal to **k**

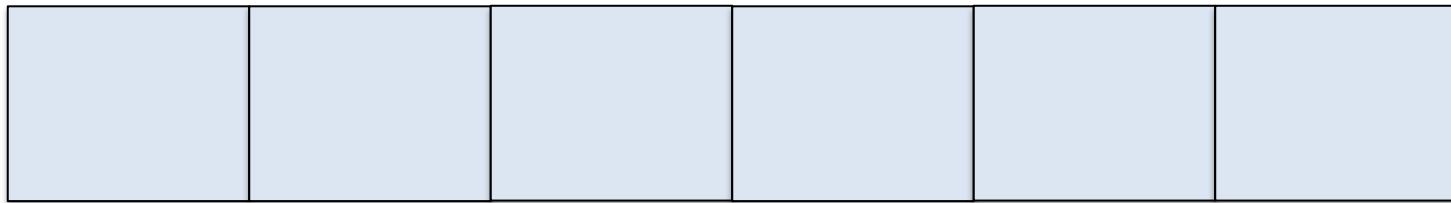


Read ID	start-pos	end-pos	Contig ID	start-pos	end-pos

Read 42	1	4	Contig 101	152	155
Read 42	130	150	Contig 61	1	21
Read 90	1	150	Contig 500	101	250

Building seed index

Seed Index



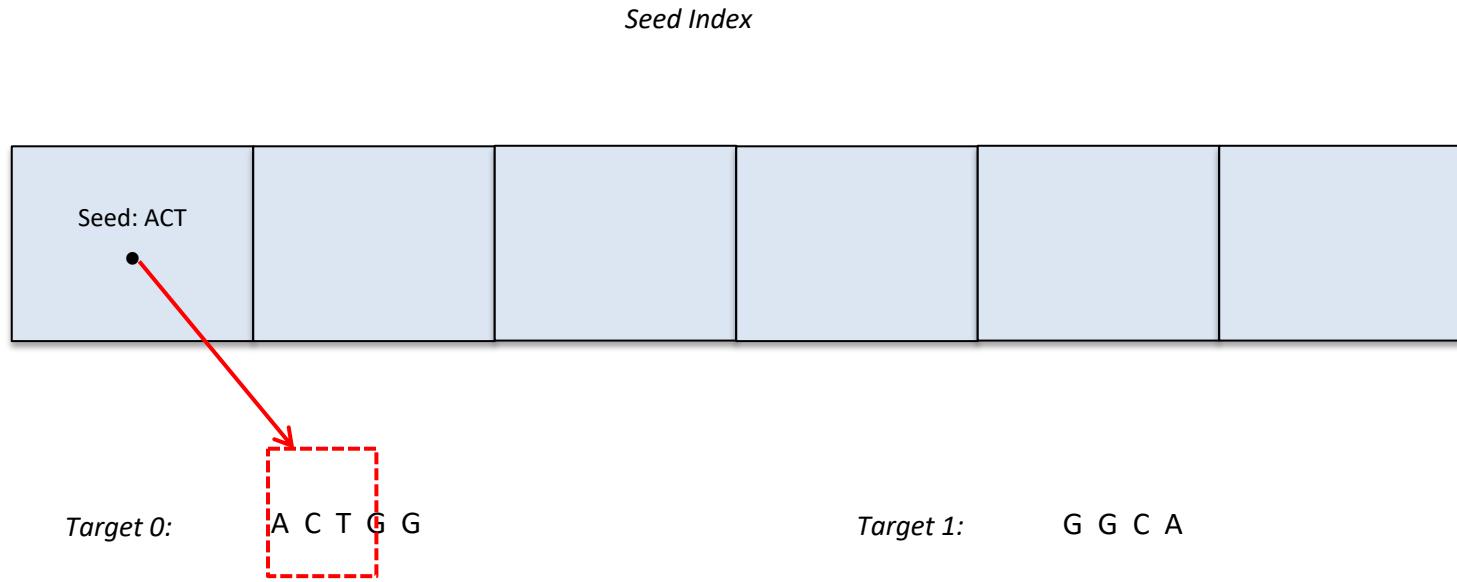
Target 0:

A C T G G

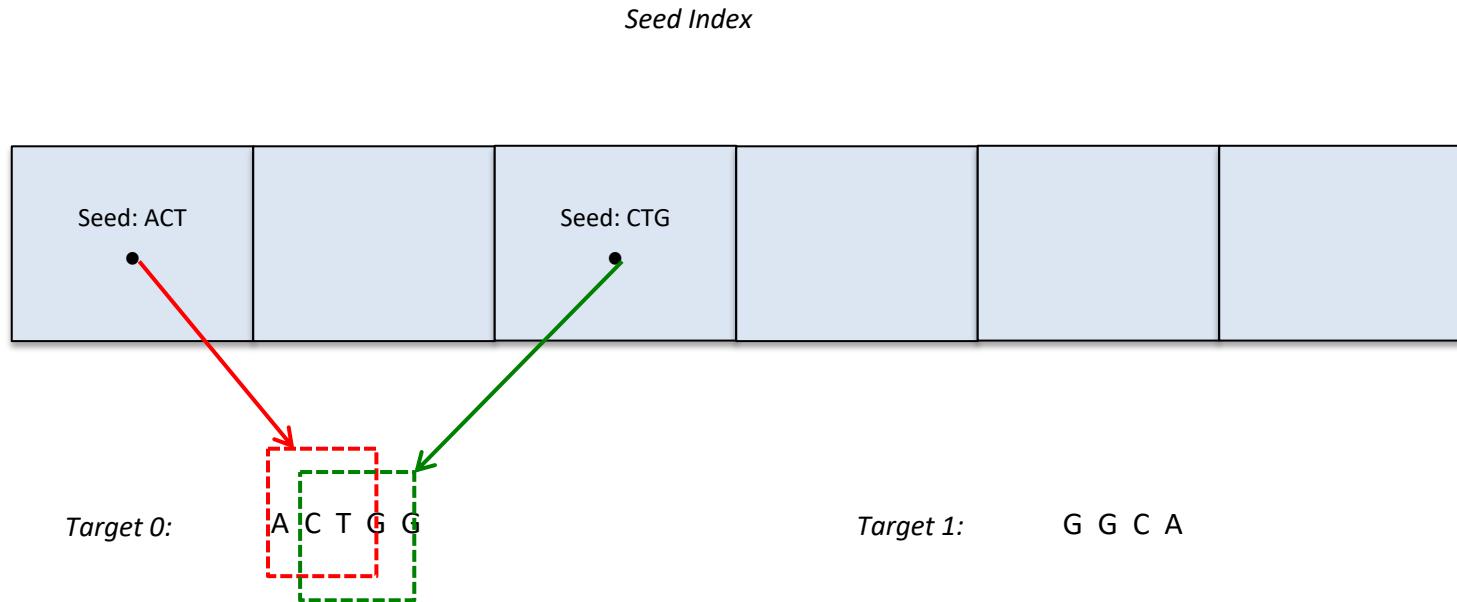
Target 1:

G G C A

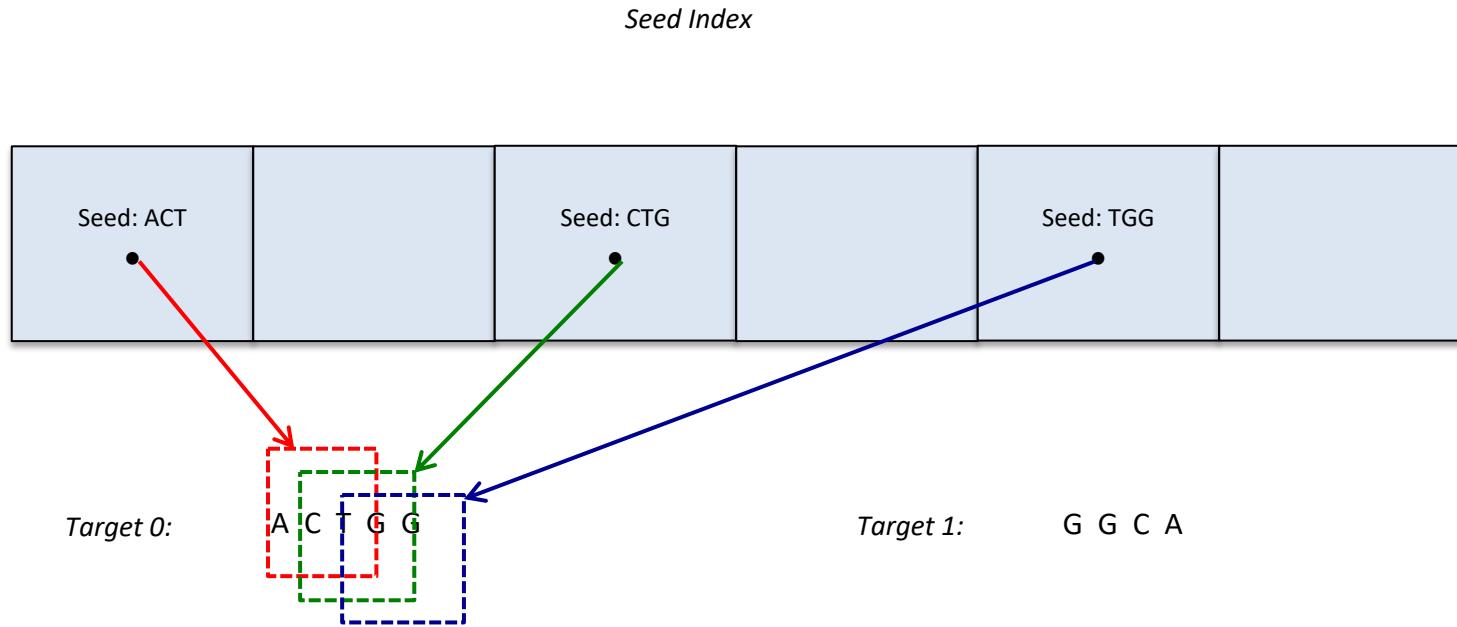
Building seed index



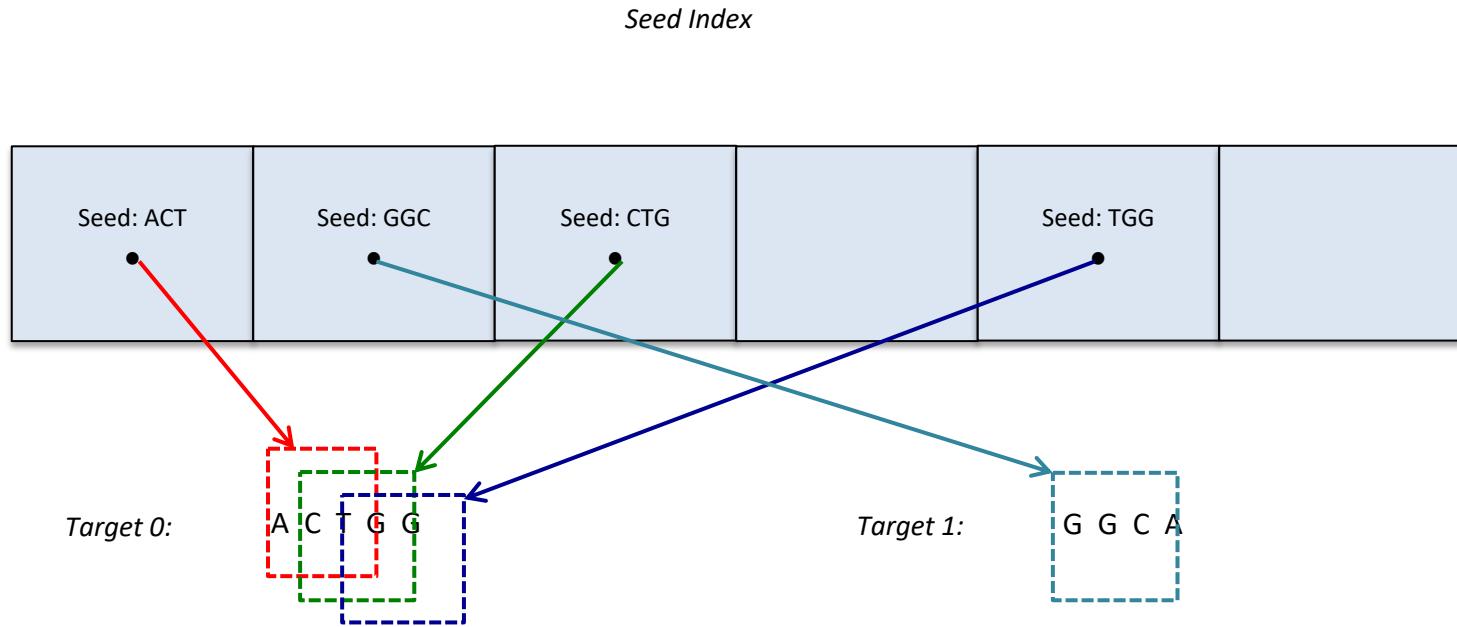
Building seed index



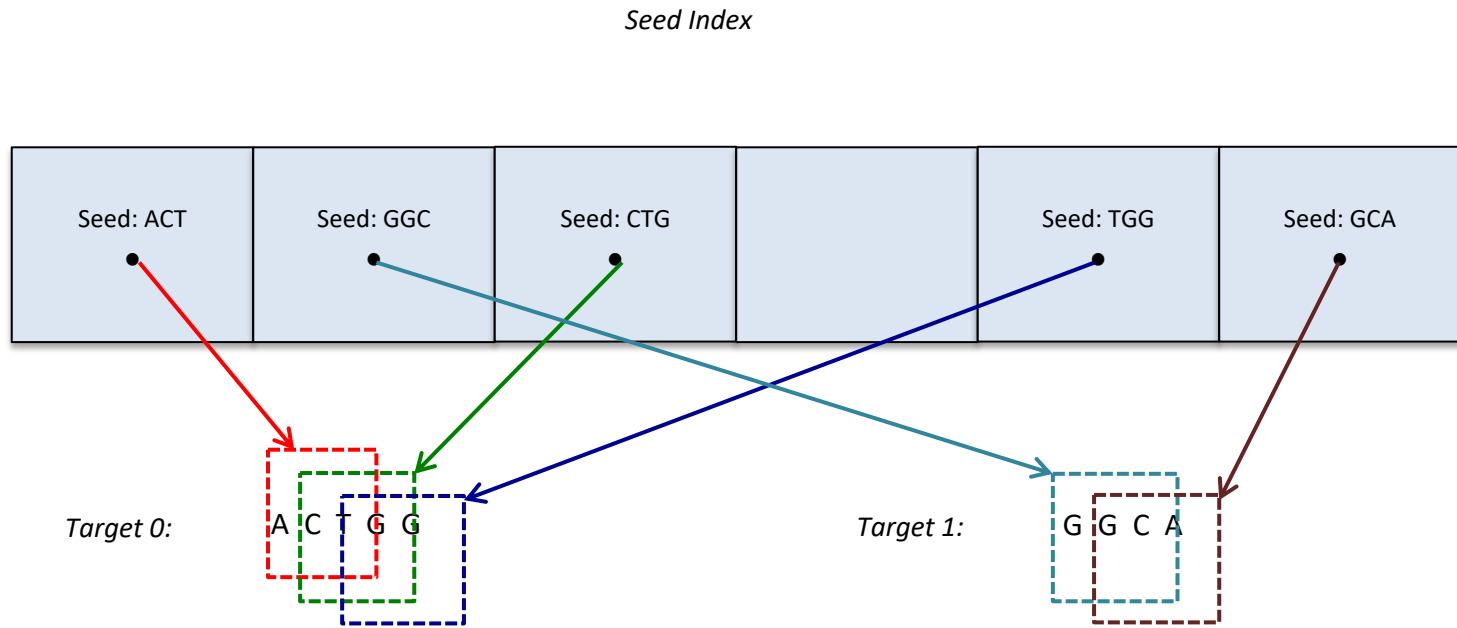
Building seed index



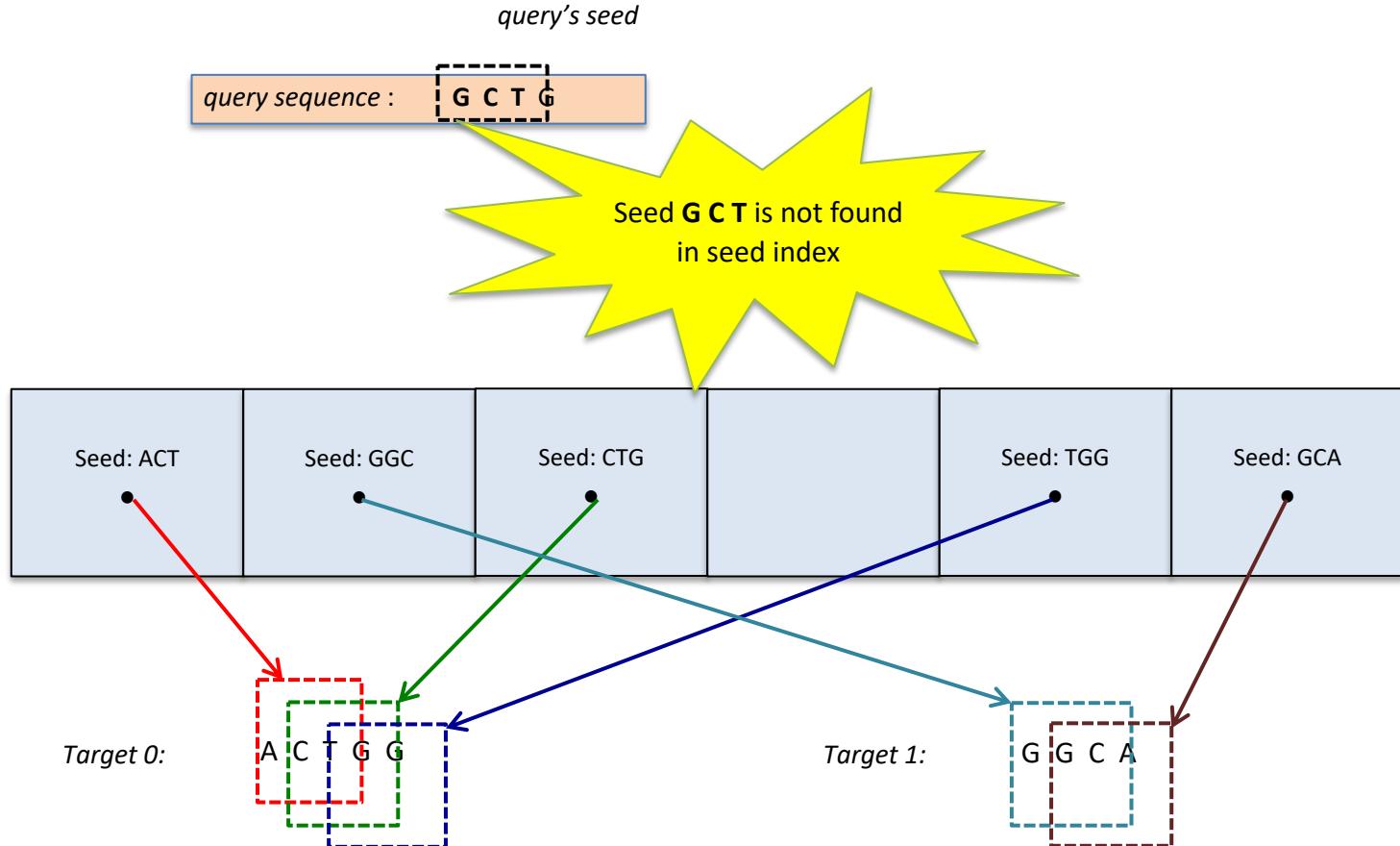
Building seed index



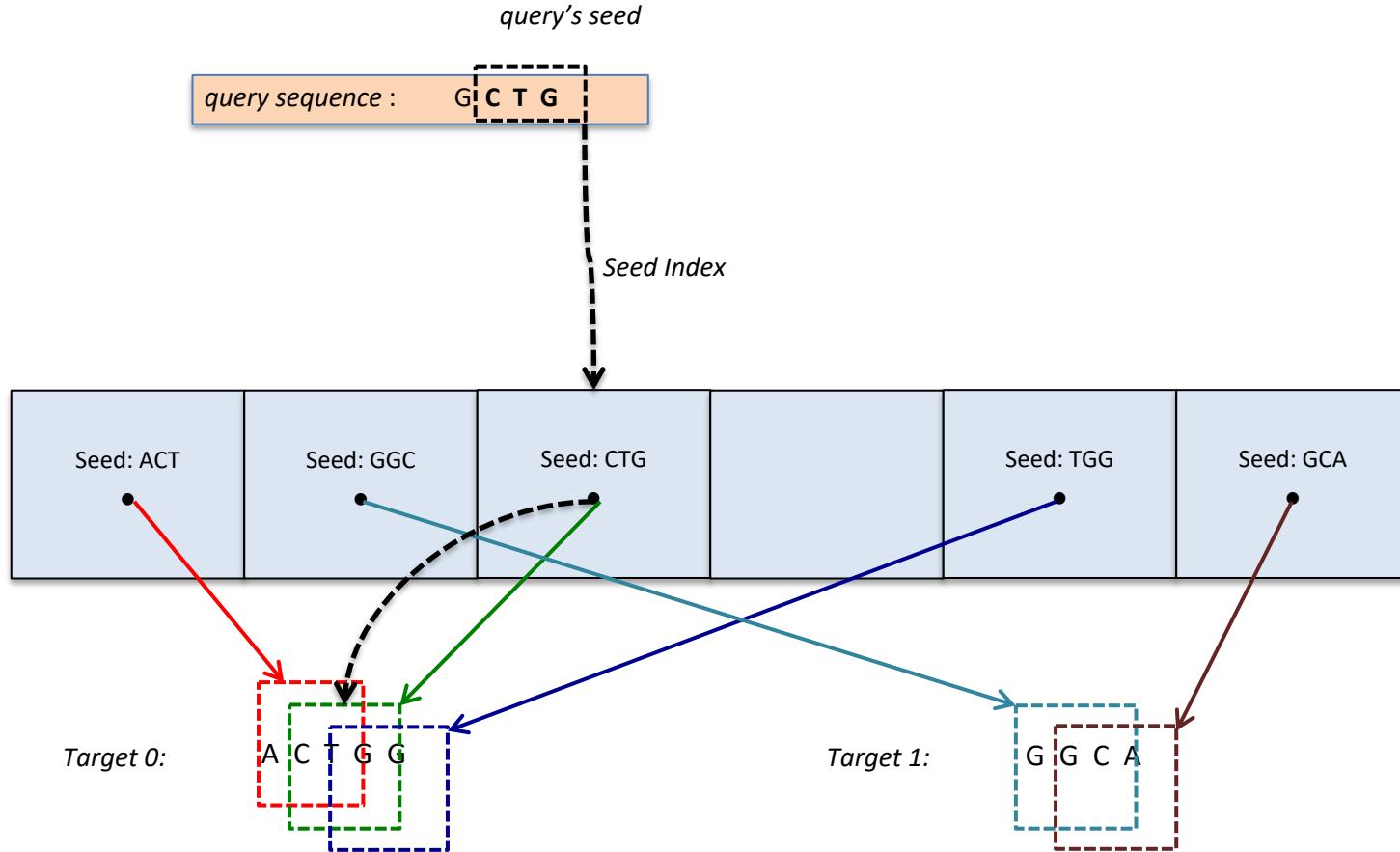
Building seed index



Lookup seed index

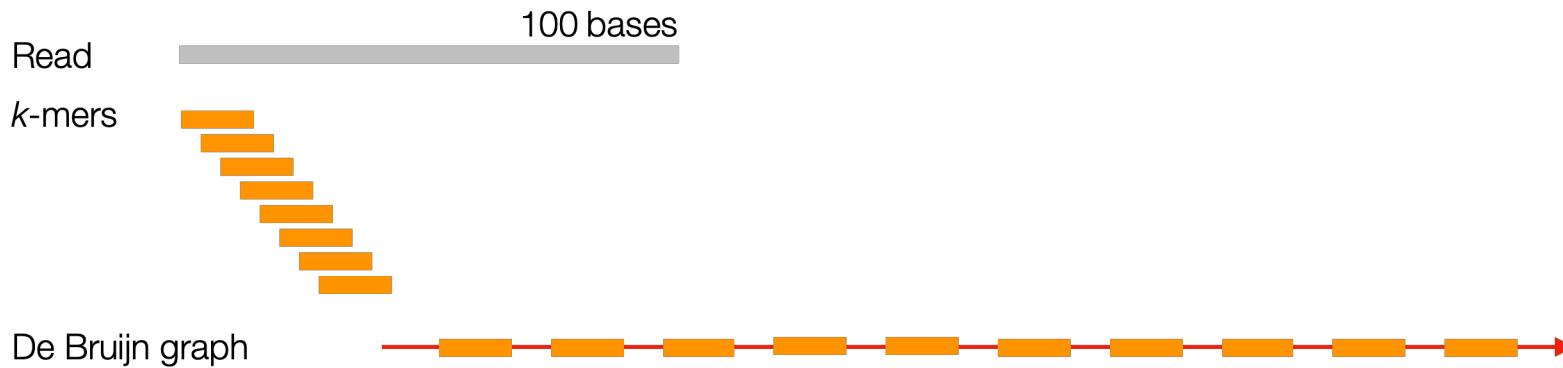


Lookup seed index

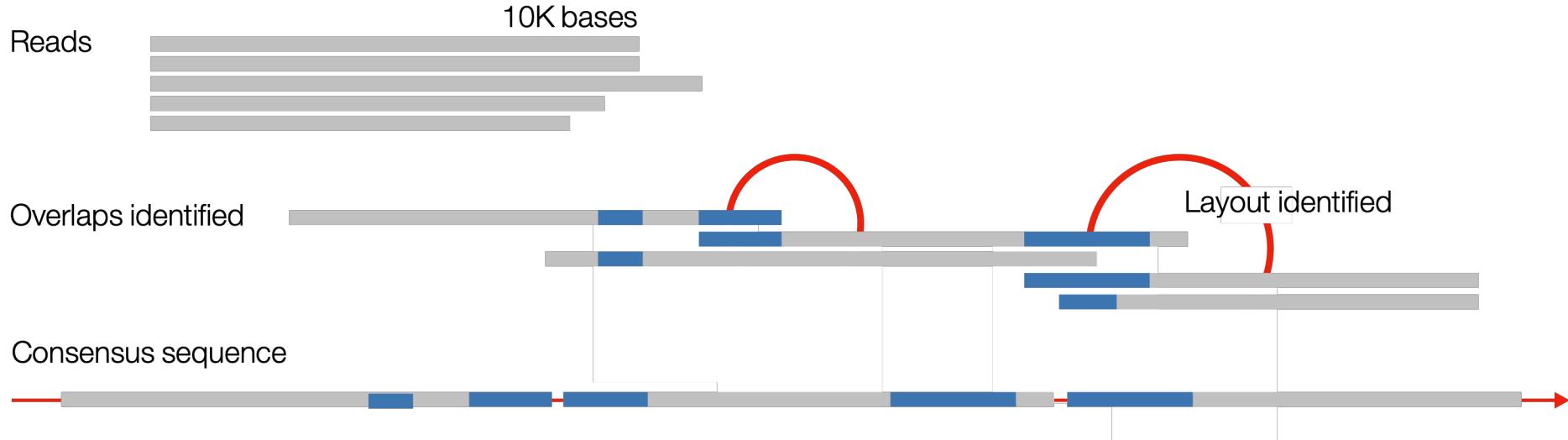


Alternative assembly paradigms

De Bruijn Graph



Overlap-Layout-Consensus

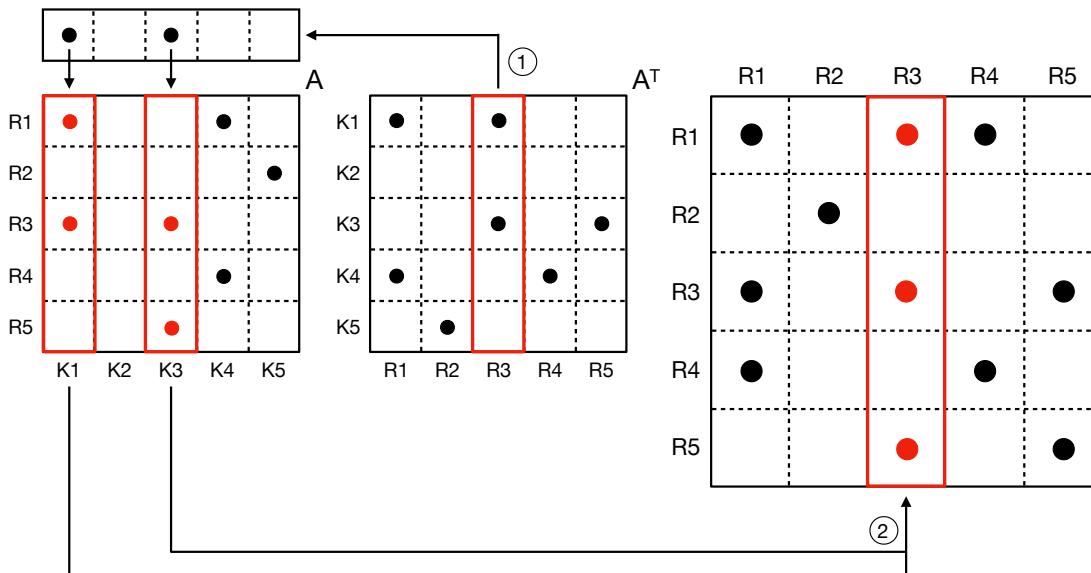


Assembly with Long Noisy Reads

- **Long reads** from PacBio and Oxford Nanopore have the potential to revolutionize de-novo assembly
- Error rates are 12-20%
- Reads are 10-100 kilo bases
- **Overlap-Consensus-Layout** paradigm is more suitable than de Bruijn graph paradigm because:
 - The overlap graph, where each vertex is a read, is smaller than de Bruijn graph for same coverage due to long reads
 - High error rates: overwhelming majority of k-mers are wrong

Candidate alignments among reads

- **Overlapping** is the most computationally expensive step in the overwhelming majority of long read assemblers.
- Imagine each read is a sample, its k-mer profile is its feature set
- Create a reads-by-kmers (sparse) matrix

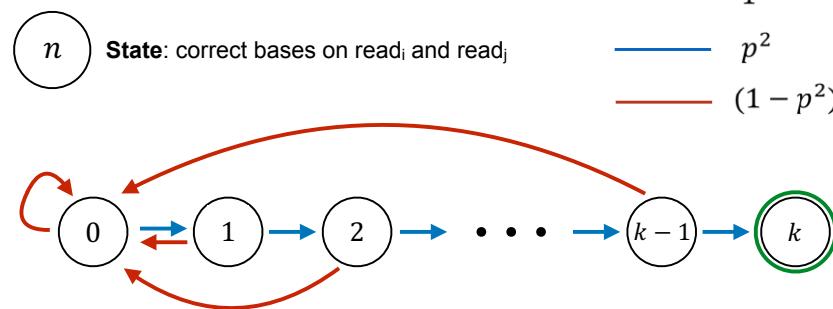


Read overlapping using shared k-mers is computing a **sparse matrix product**, for which we know good algorithms and implementations

BELLA: Berkeley Long-read to Long-read Aligner and Overlapper

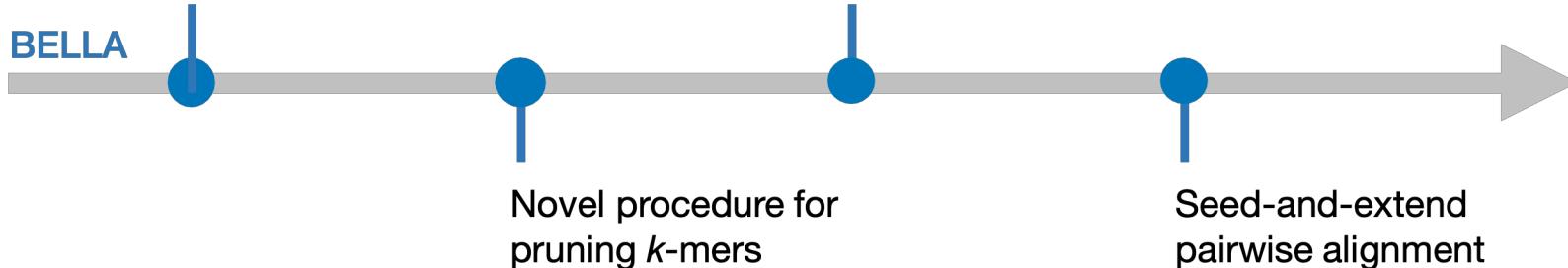
Number of states: $k + 1$

Legend:



- How to choose the right set of k-mers, otherwise there are too many of them?
- How to use alignment score to tell true alignments from false positives?

Feasibility of
a k -mer seed based approach



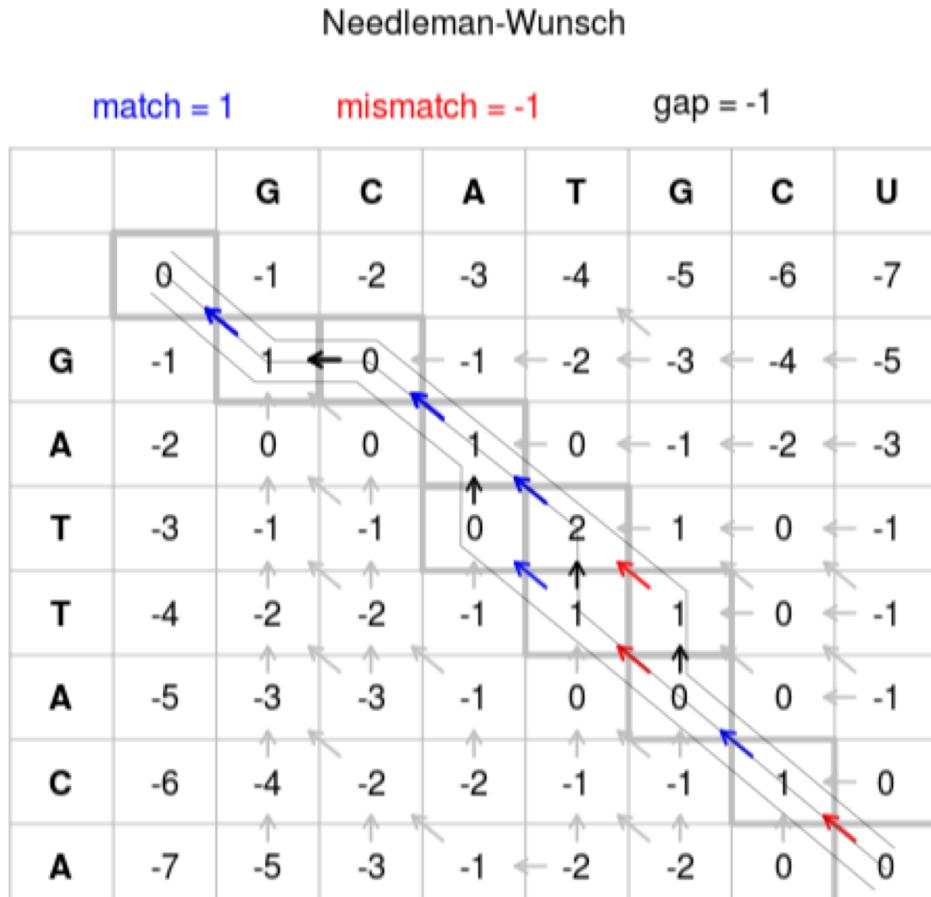
Guidi G, Ellis M, Rokhsar D, Yelick K, Buluç A. BELLA: Berkeley Efficient Long-Read to Long-Read Aligner and Overlapper. bioRxiv 464420; doi: <https://doi.org/10.1101/464420>

Outline

- Genome (and metagenome) assembly problems
- Indexing with hash tables
- **Pairwise sequence alignment**
- Indexing with suffix arrays

Pairwise sequence alignment

1. Finding globally best alignment



$$M(i,j) = \max \begin{cases} M(i-1, j) + \text{gap} \\ M(i, j-1) + \text{gap} \\ M(i-1, j-1) + \text{score}(s_i, t_j) \end{cases}$$

Two input sequences:
 $S=s_1s_2\dots s_m$, $T=t_1t_2\dots t_n$

Various ordering options to fill DP (dynamic programming) table as long as dependencies are met

- row-by-row,
- column-by-column,
- diagonal-by-diagonal...

Complexity: $O(mn)$

Pairwise sequence alignment

2. Finding locally best alignment

$$M(i,j) = \max \begin{cases} M(i-1, j) + \text{gap} \\ M(i, j-1) + \text{gap} \\ M(i-1, j-1) + \text{score}(s_i, t_j) \\ 0 \end{cases}$$

Smith-Waterman Algorithm

Complexity: $O(mn)$

Instead of tracing back from bottom right, one traces back from the highest score

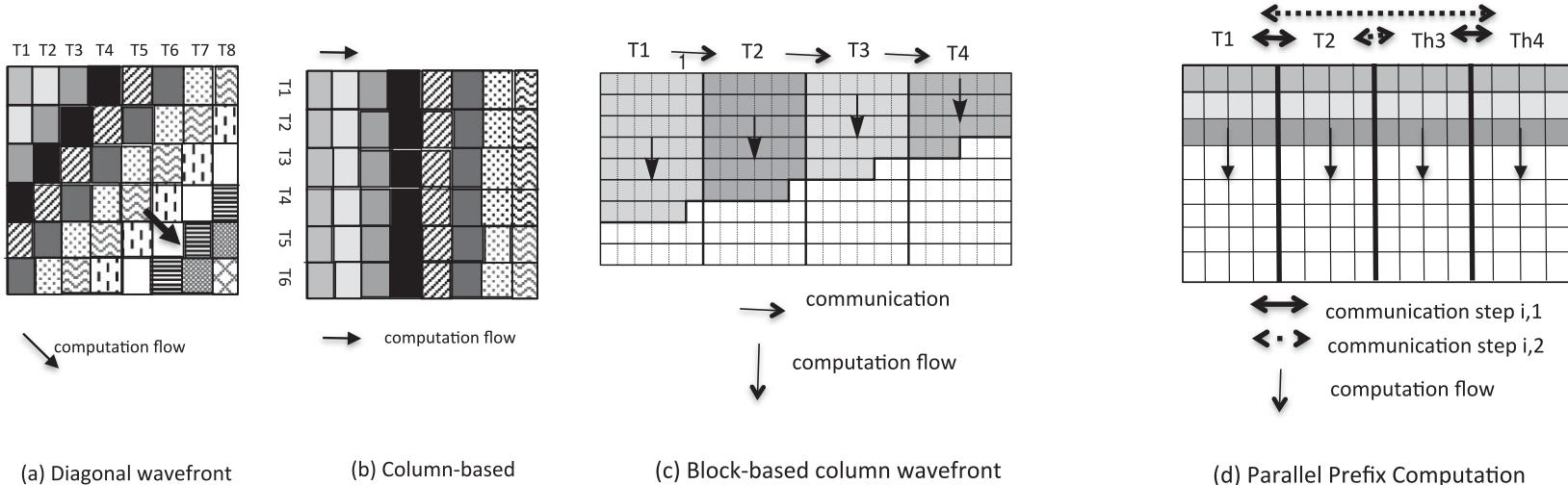
Initialize the scoring matrix

	T	G	T	T	A	C	G	G
T	0	0	0	0	0	0	0	0
G	0							
G	0							
T	0							
T	0							
G	0							
A	0							
C	0							
T	0							
A	0							

Substitution matrix: $S(a_i, b_j) = \begin{cases} +3, & a_i = b_j \\ -3, & a_i \neq b_j \end{cases}$

Gap penalty: $W_k = kW_1$
 $W_1 = 2$

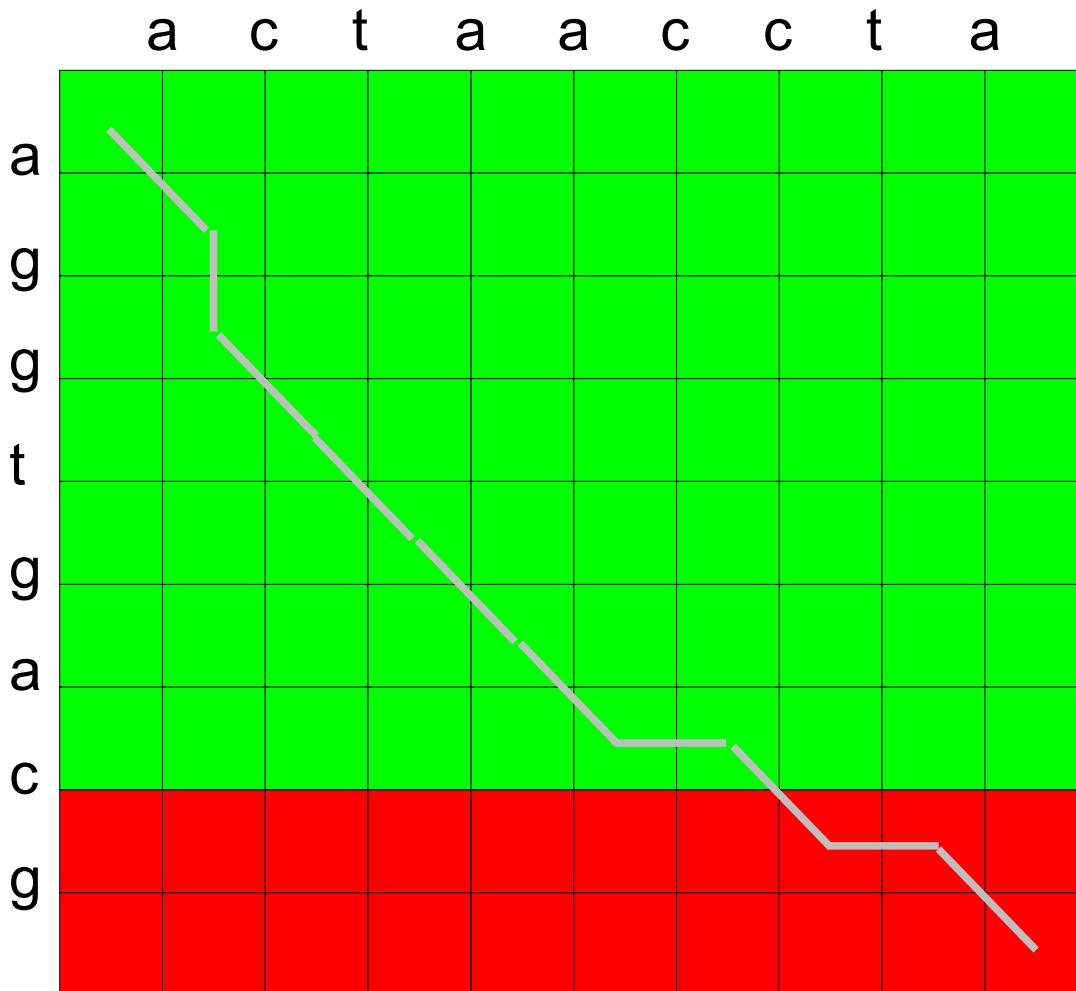
Parallel sequence alignment



- Several ways of exploiting **fine-grained parallelism** in sequence alignment
- Need long(ish) sequences for exposing enough parallelism

Sandes EF, Boukerche A, Melo AC. Parallel optimal pairwise biological sequence comparison: algorithms, platforms, and classification. ACM Computing Surveys (CSUR). 2016 2;48(4):63.

Linear space – Hirschberg's idea



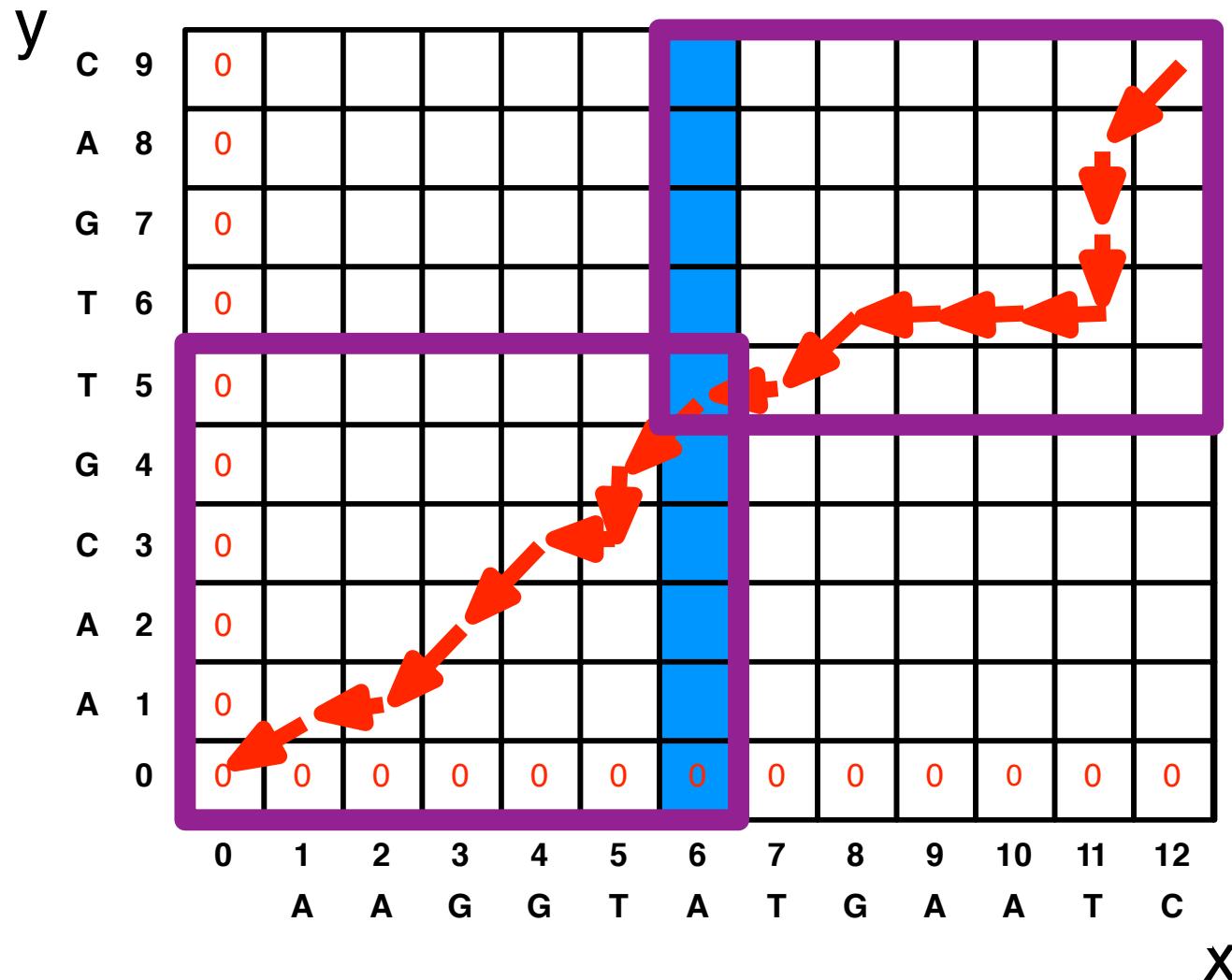
Hirschberg's idea:

Find “middle edge” (i.e. “middle column”) of an optimal alignment, split into two subproblems and recurse ...

Time: nm

Can I really find the middle edge in $O(mn)$ time in linear space?

The Best Path Uses Some Cell in the Middle Column



First Attempt At Space Efficient Alignment

In the optimal alignment, the first $n/2$ characters of x are aligned with the first q characters of y for some q .

12345678
$x = \text{ACGTACTG}$
$y = \underbrace{\text{A-GT}}_{q=3} \text{CTG}$
$q = 3$

We don't know q , so we have to try *all* possible q .

```
ArrowPath := []
def Align(x, y):
    n := |x|; m := |y|
    if n or m ≤ 2: use standard alignment
    for q := 0..m:
        v1 := AlignValue(x[1..n/2], y[1..q])
        v2 := AlignValue(x[n/2+1..n], y[q+1..m])
        if v1 + v2 < best: bestq = q; best = v1 + v2
    Add (n/2, bestq) to ArrowPath
    Align(x[1..n/2], y[1..bestq])
    Align(x[n/2+1..n], y[bestq+1..m])
```

$O(n)$ or $O(m)$ space

$O(n+m)$ space

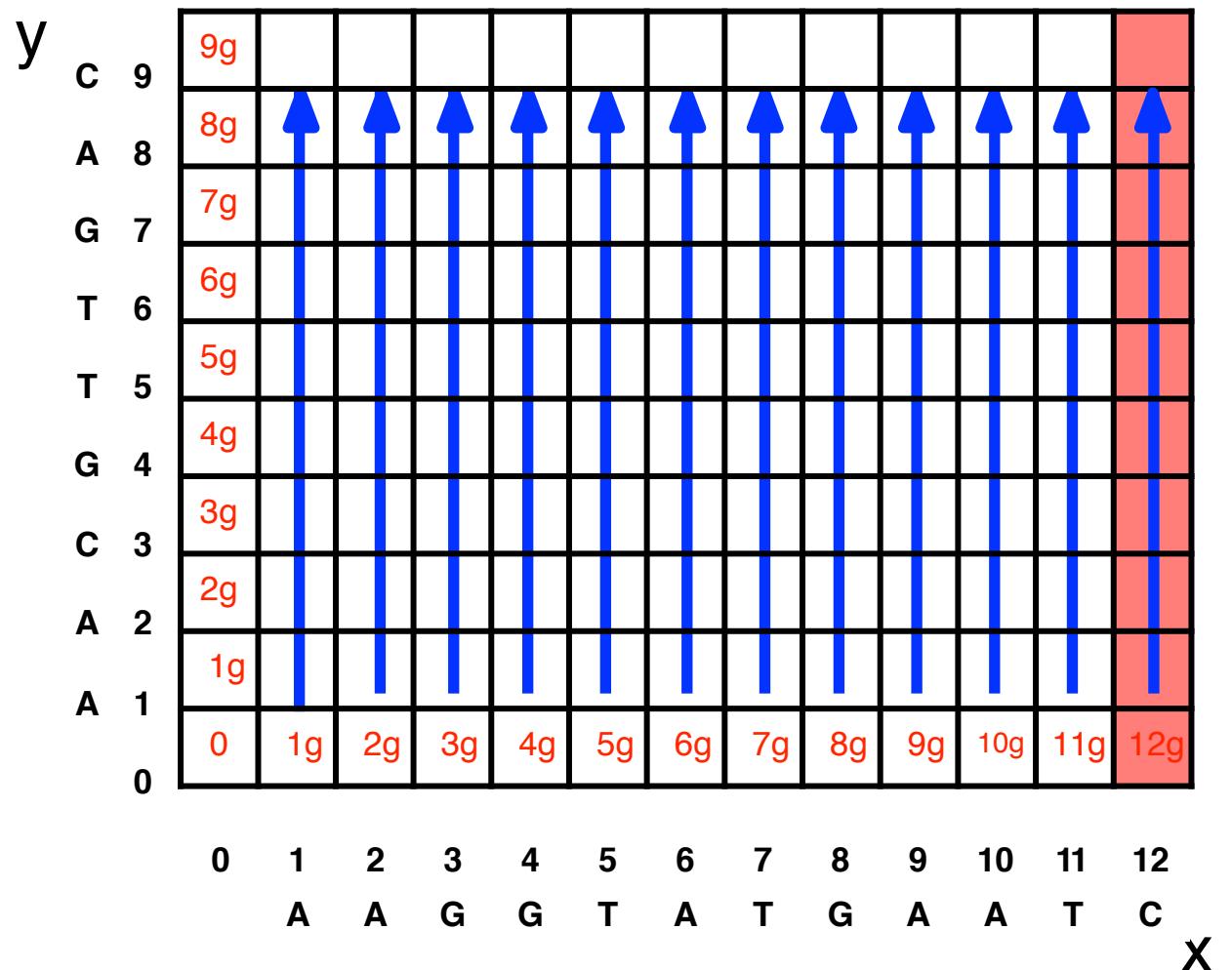
$O(n+m)$ space

find the q that minimizes
the cost of the alignment

Problem

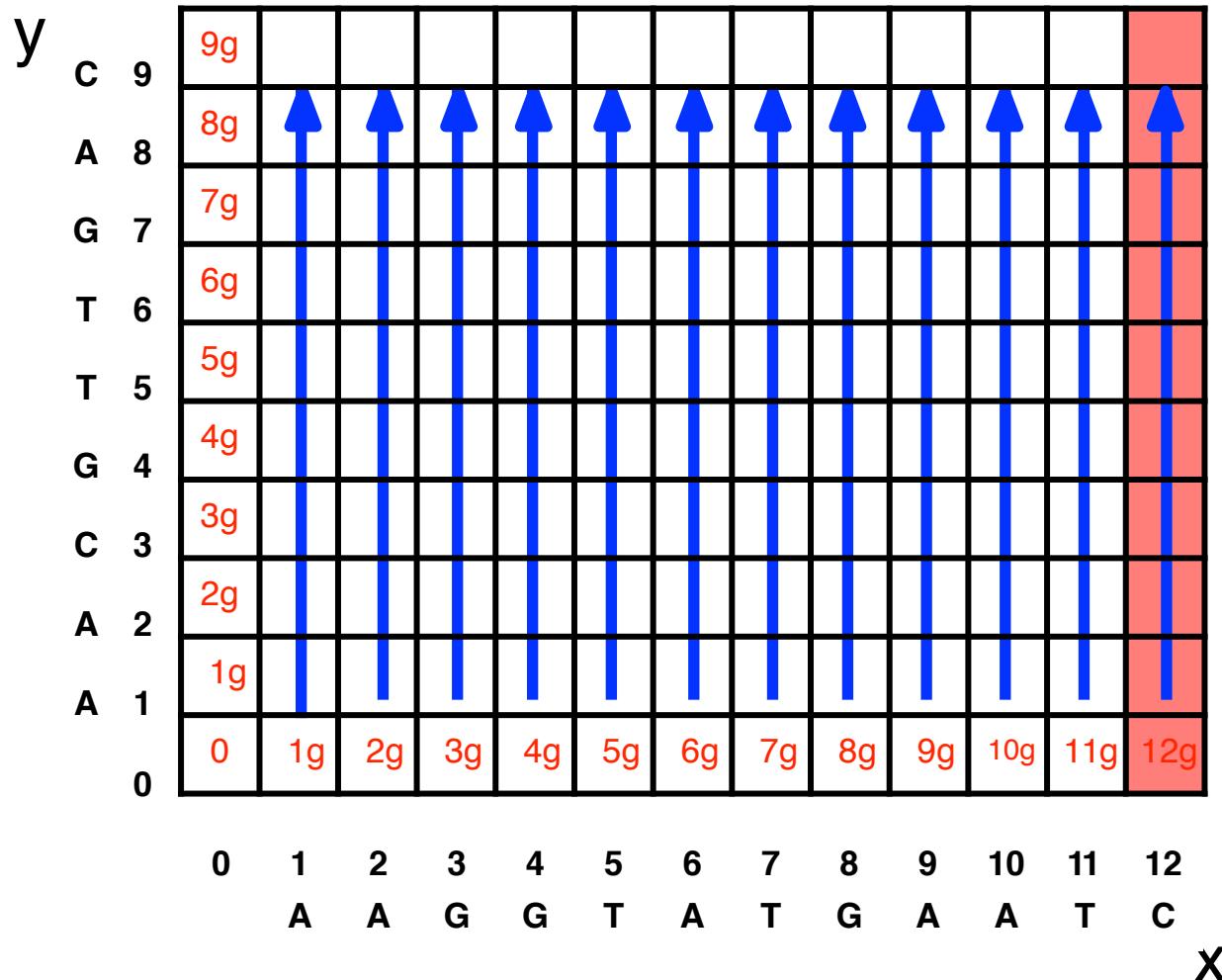
- This works in linear space.
- BUT: not in $O(nm)$ time.
- It's too expensive to solve all those AlignValue problems in the **for** loop.
- Define:
 - **AllYPrefixCosts**(x, i, y) = returns an array of the scores of optimal alignments between $x[1..i]$ and all prefixes of y .
 - **AllYSuffixCosts**(x, i, y) = returns an array of the scores of optimal alignments between $x[i..n]$ and all suffixes of y
 - These are implemented as described in previous slides by returning the last row or last column of the DP matrix.

Fill in the matrix by columns...



What is this
column?

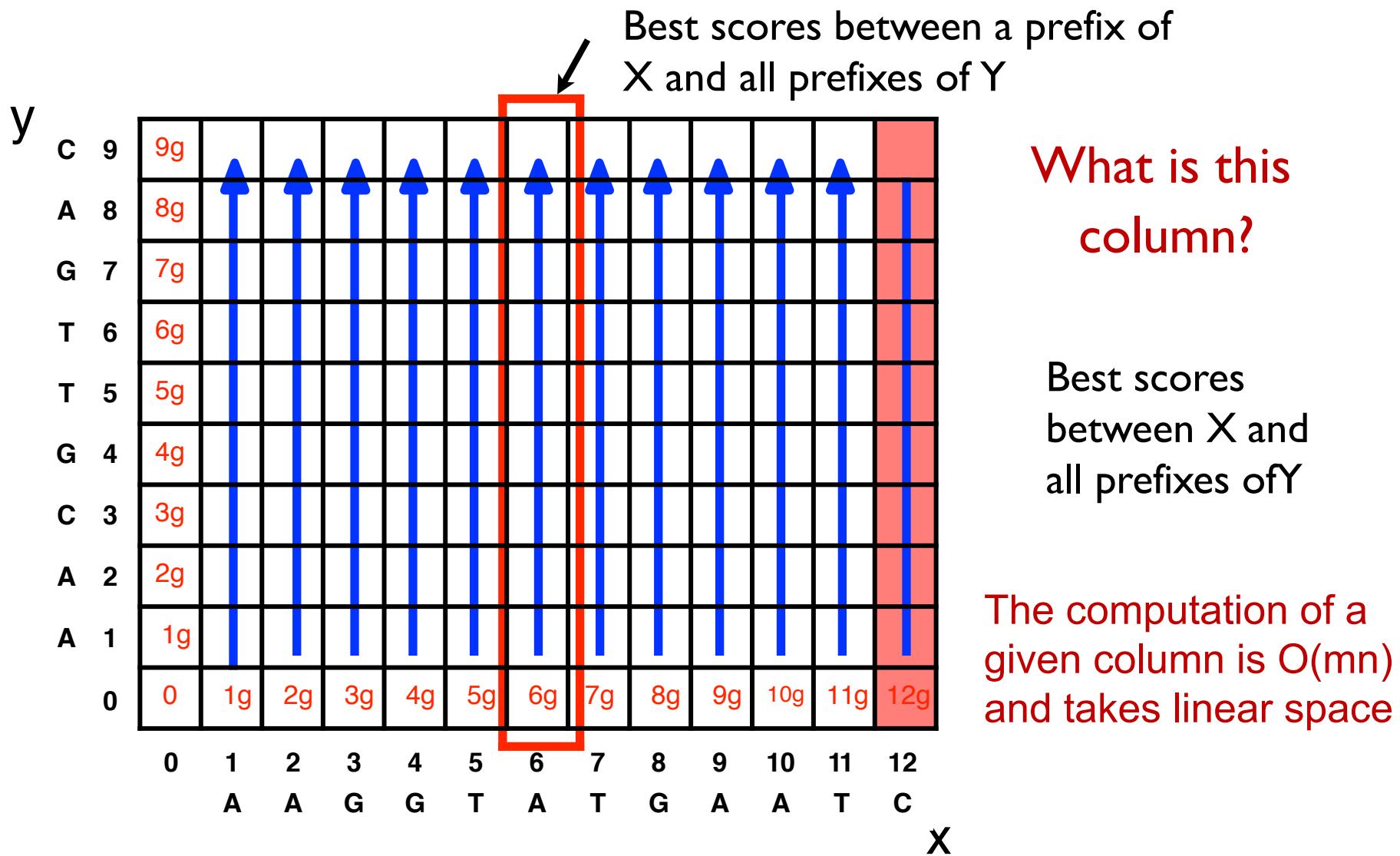
Fill in the matrix by columns...



What is this column?

Best scores
between X and
all prefixes of Y

Fill in the matrix by columns...



Space Efficient Alignment

We still try all possible q , but we use the fact that we can compute the cost between a given prefix and *all* suffixes in linear space.

12345678
x = ACGTACTG
y = $\underbrace{A-GT-CTG}_{q=3}$

ArrowPath := []

```
def Align(x, y):  
    n := |x|; m := |y|  
    if n or m ≤ 2: use standard alignment
```

$O(n)$ or $O(m)$ space

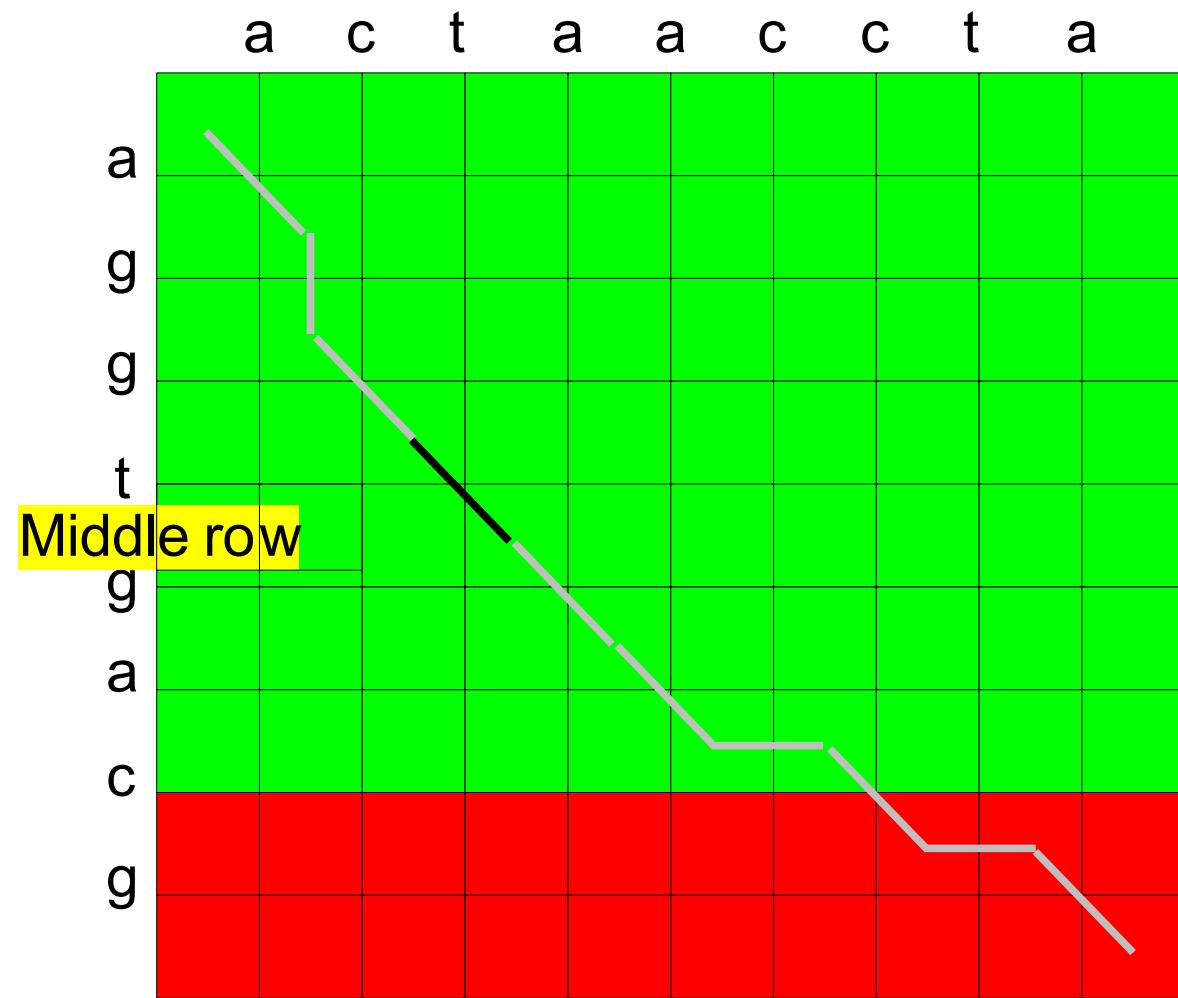
```
YPrefix := AllYPrefixCosts(x, n/2, y)  
YSuffix := AllYSuffixCosts(x, n/2+1, y) }  $O(n+m)$  space
```

```
for q := 0..m:  
    cost = YPrefix[q] + YSuffix[q+1]  
    if cost < best: bestq = q; best = cost
```

— find the q that minimizes the cost of the alignment, using the costs of aligning X to prefixes and suffixes of Y

```
Add (n/2, bestq) to ArrowPath  
Align(x[1..n/2], y[1..bestq])  
Align(x[n/2+1..n], y[bestq+1..m])
```

Linear space – Hirschberg's idea



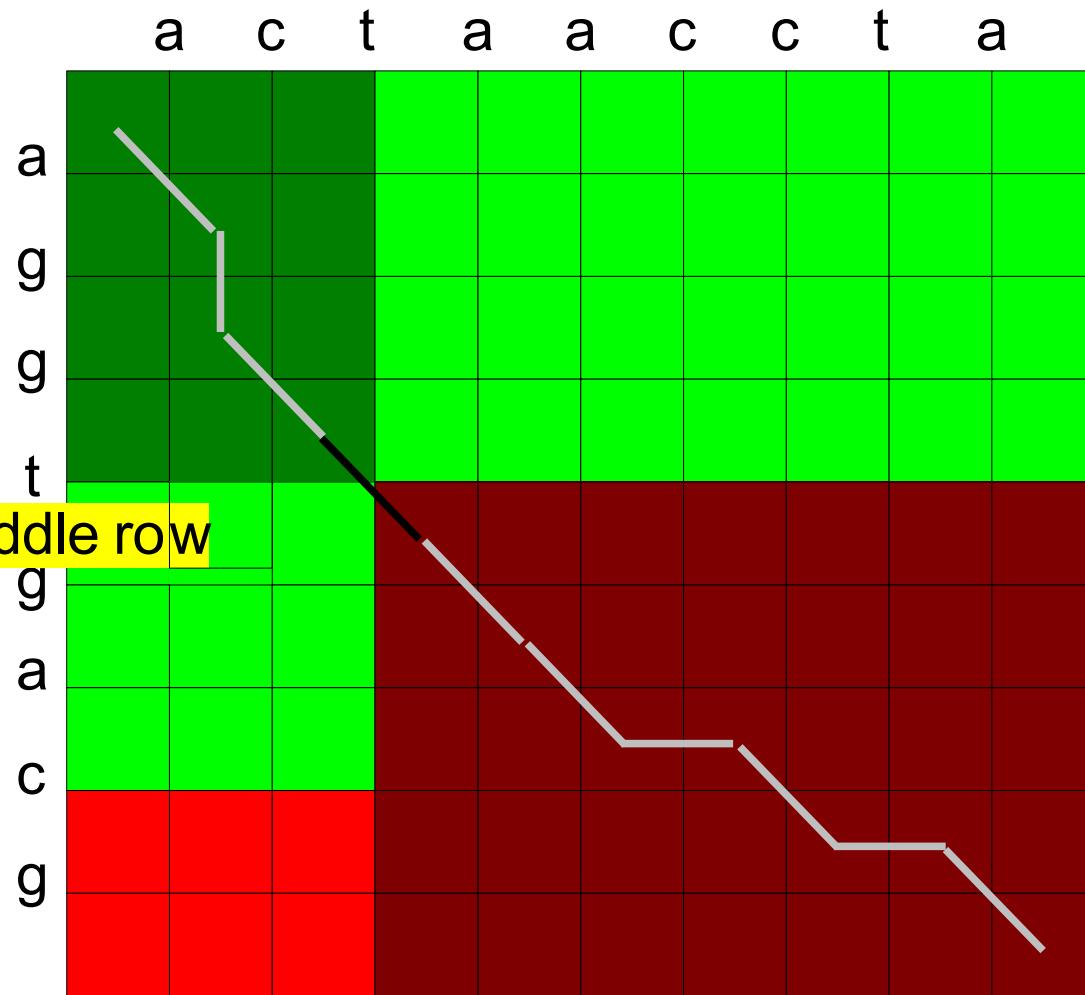
Hirschberg's idea:

Find “middle edge” (i.e. “middle column”) of an optimal alignment, split into two subproblems and recurse ...

Time:

nm

Linear space – Hirschberg's idea



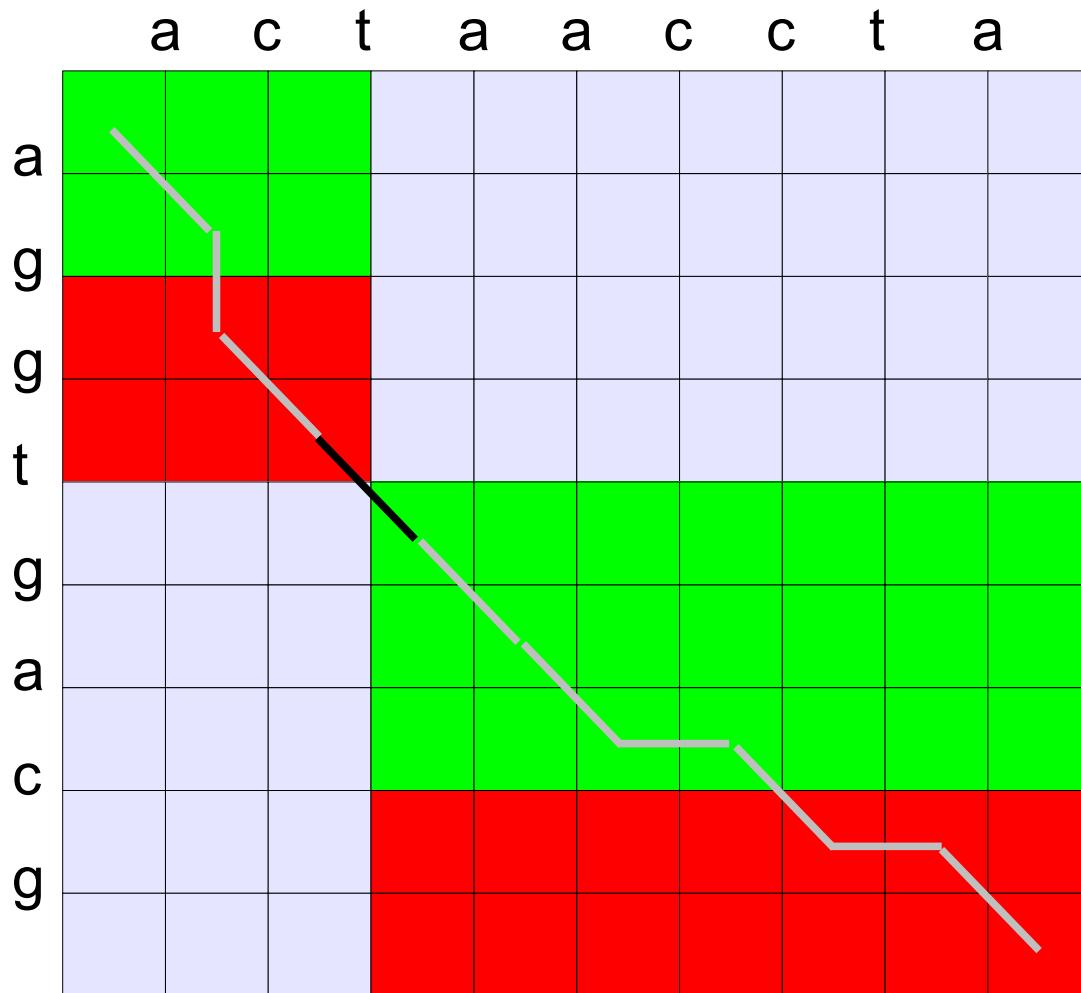
Hirschberg's idea:

Find “middle edge” (i.e. “middle column”) of an optimal alignment, split into two subproblems and recurse ...

Time:

nm

Linear space – Hirschberg's idea



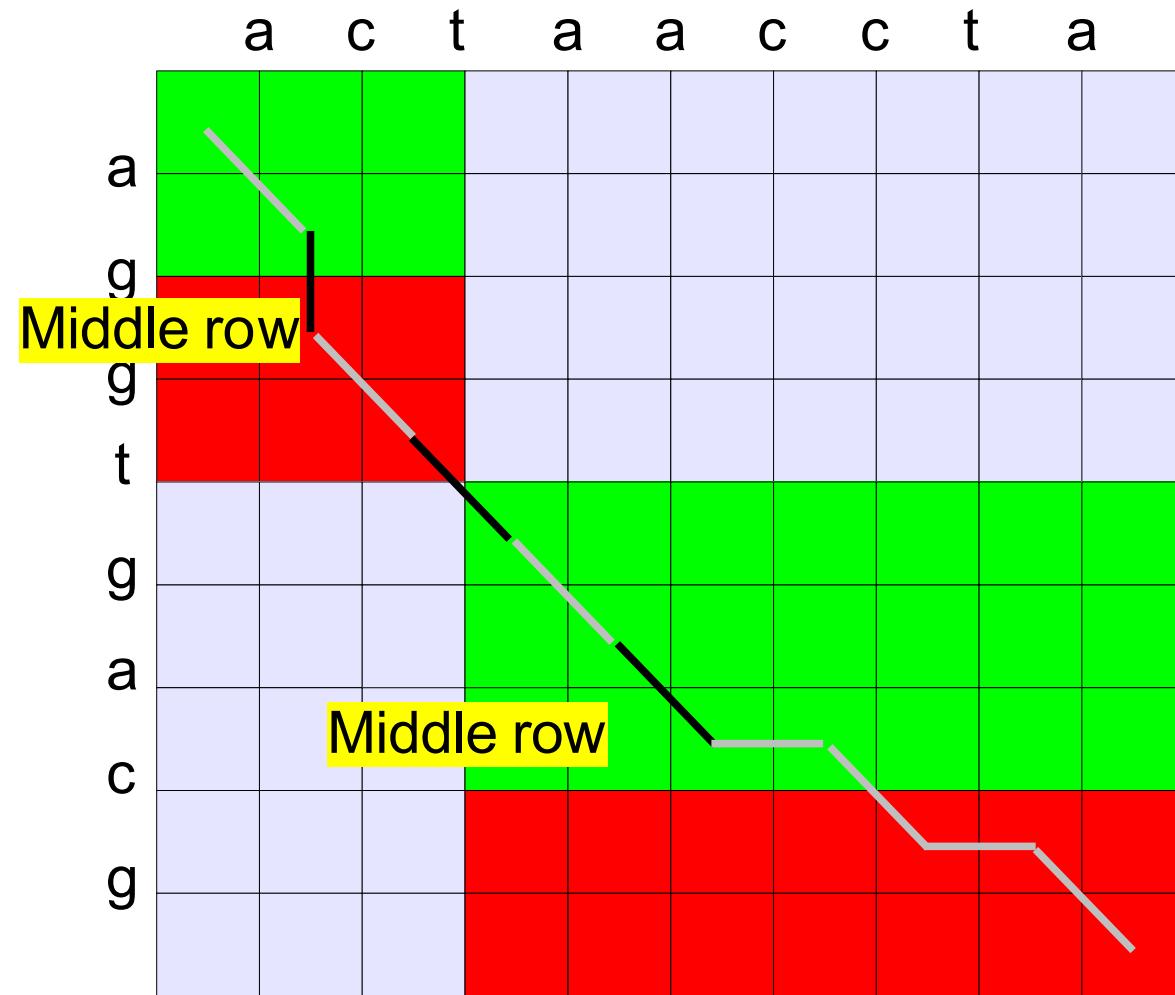
Hirschberg's idea:

Find “middle edge” (i.e. “middle column”) of an optimal alignment, split into two subproblems and recurse ...

Time:

$$nm + (n/2)m$$

Linear space – Hirschberg's idea



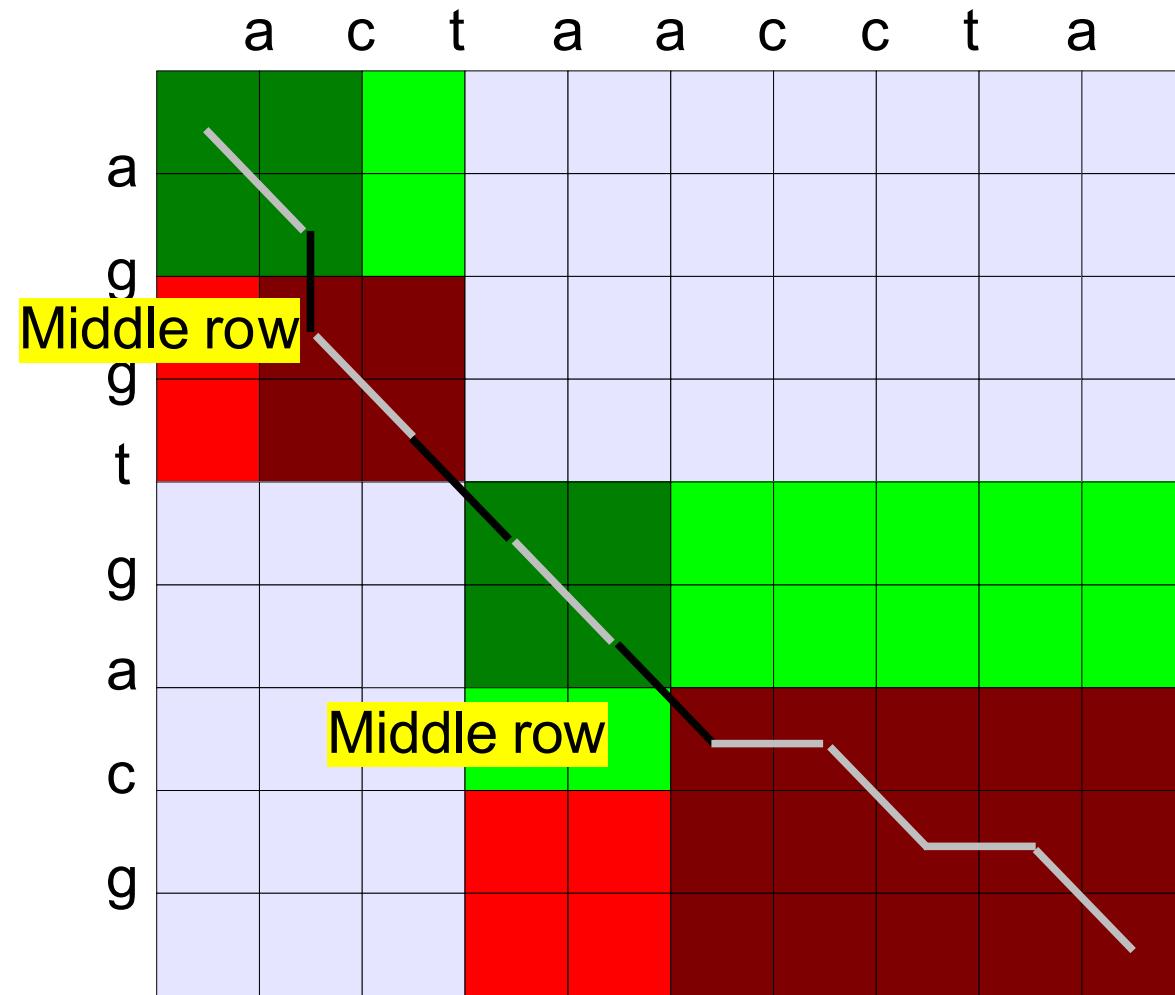
Hirschberg's idea:

Find “middle edge” (i.e. “middle column”) of an optimal alignment, split into two subproblems and recurse ...

Time:

$$nm + (n/2)m$$

Linear space – Hirschberg's idea



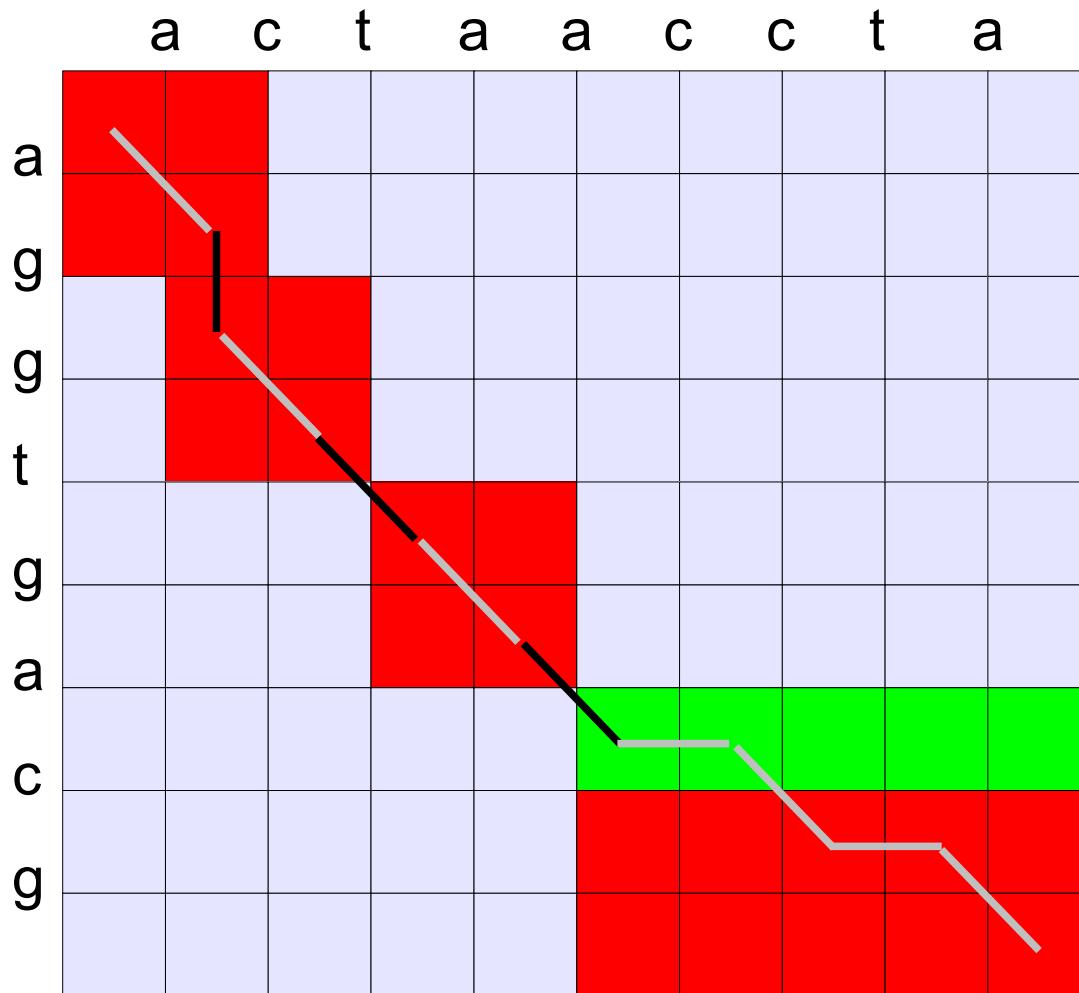
Hirschberg's idea:

Find “middle edge” (i.e. “middle column”) of an optimal alignment, split into two subproblems and recurse ...

Time:

$$nm + (n/2)m$$

Linear space – Hirschberg's idea



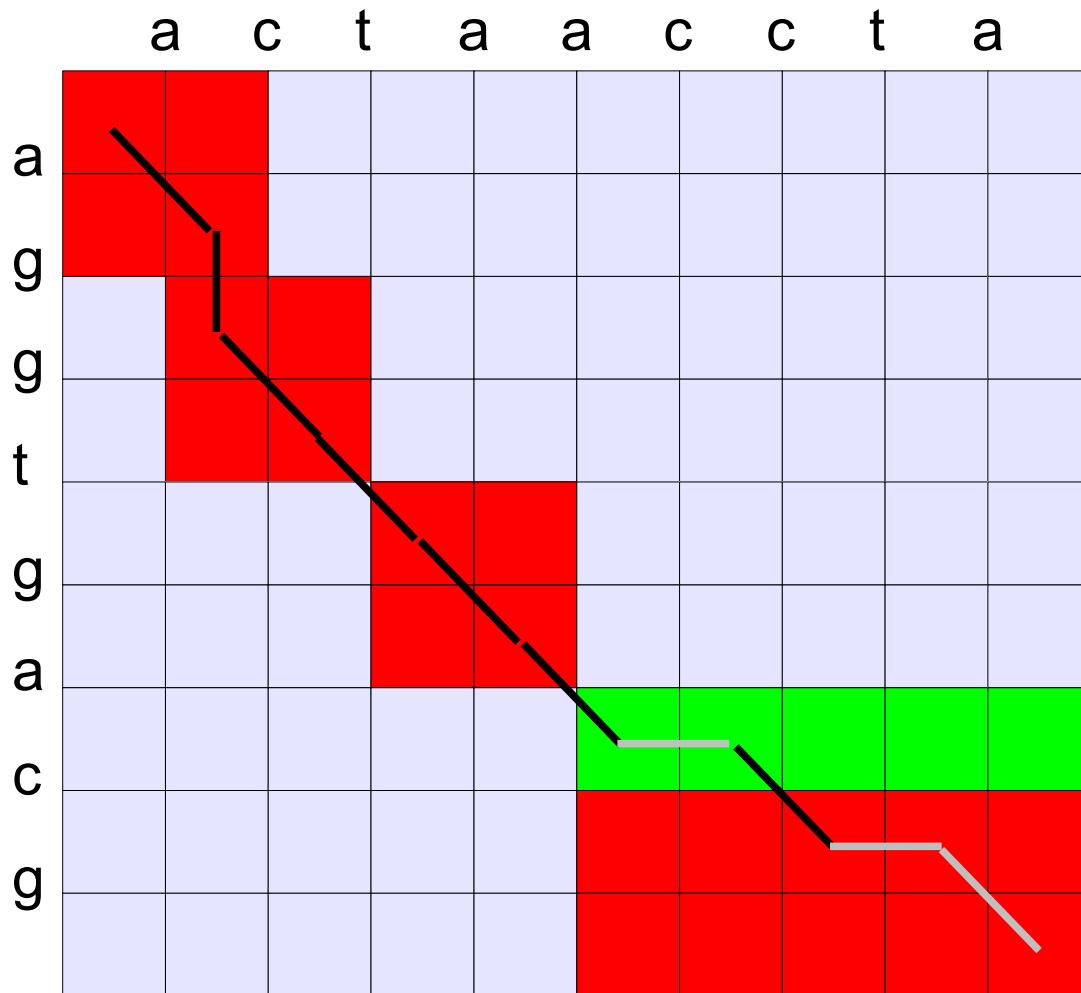
Hirschberg's idea:

Find “middle edge” (i.e. “middle column”) of an optimal alignment, split into two subproblems and recurse ...

Time:

$$nm + (n/2)m + (n/4)m$$

Linear space – Hirschberg's idea



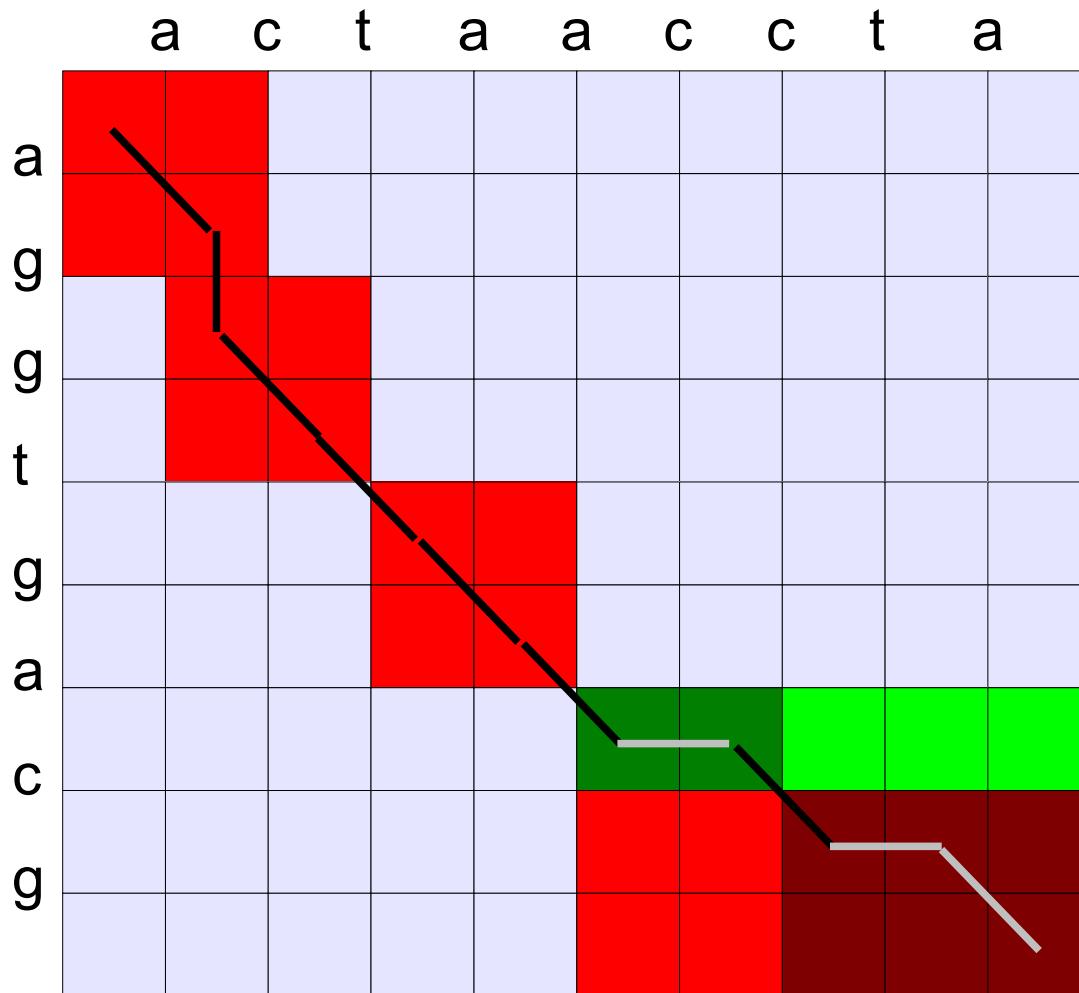
Hirschberg's idea:

Find “middle edge” (i.e. “middle column”) of an optimal alignment, split into two subproblems and recurse ...

Time:

$$nm + (n/2)m + (n/4)m$$

Linear space – Hirschberg's idea



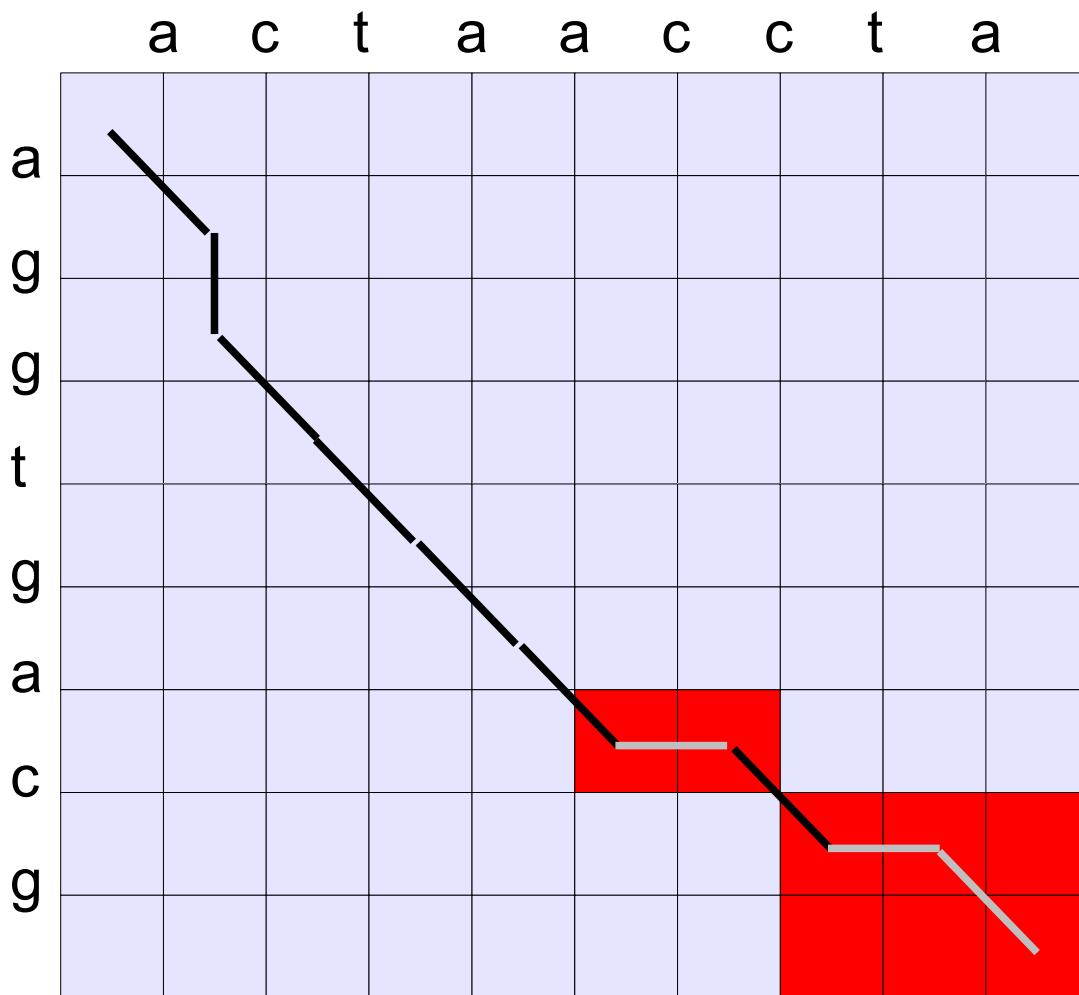
Hirschberg's idea:

Find “middle edge” (i.e. “middle column”) of an optimal alignment, split into two subproblems and recurse ...

Time:

$$nm + (n/2)m + (n/4)m$$

Linear space – Hirschberg's idea



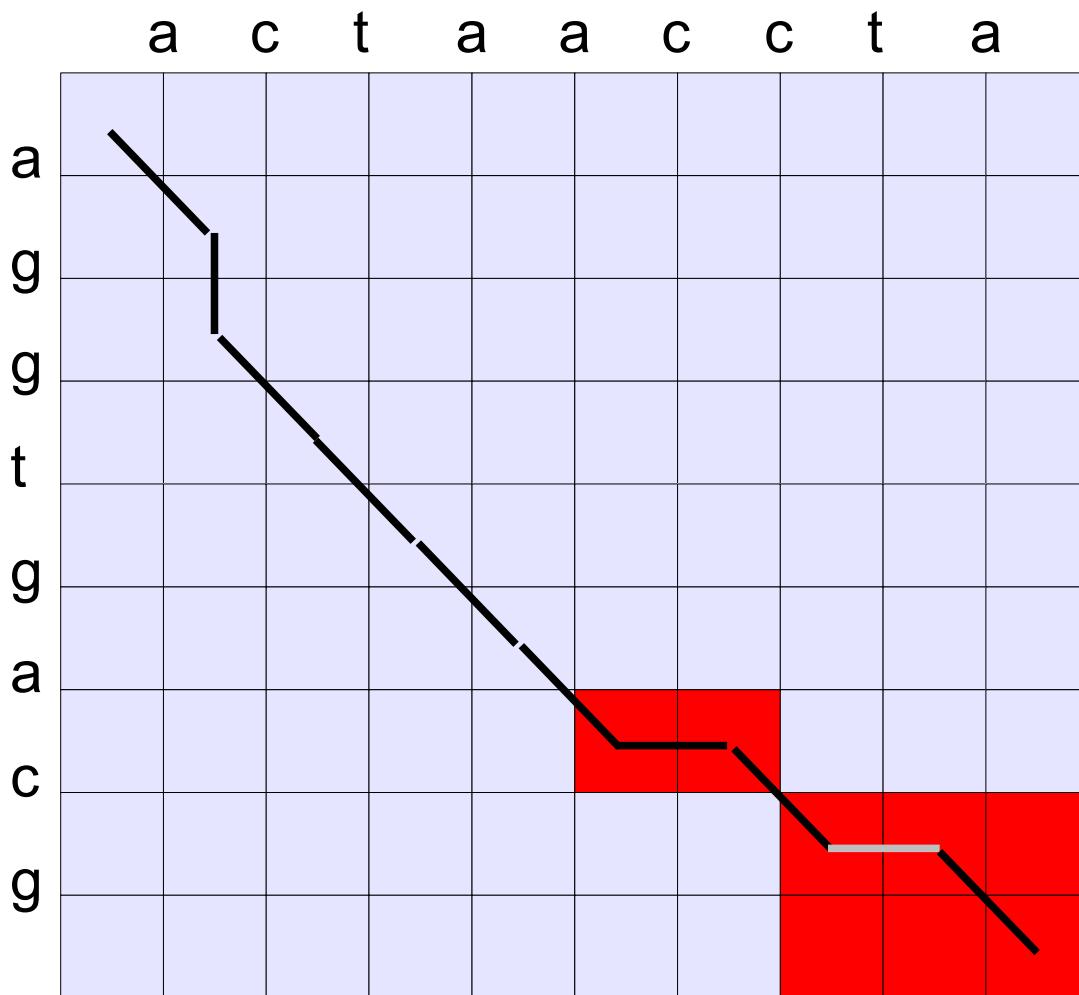
Hirschberg's idea:

Find “middle edge” (i.e. “middle column”) of an optimal alignment, split into two subproblems and recurse ...

Time:

$nm +$
 $(n / 2)m +$
 $(n / 4)m +$
 $(n / 8)m$

Linear space – Hirschberg's idea



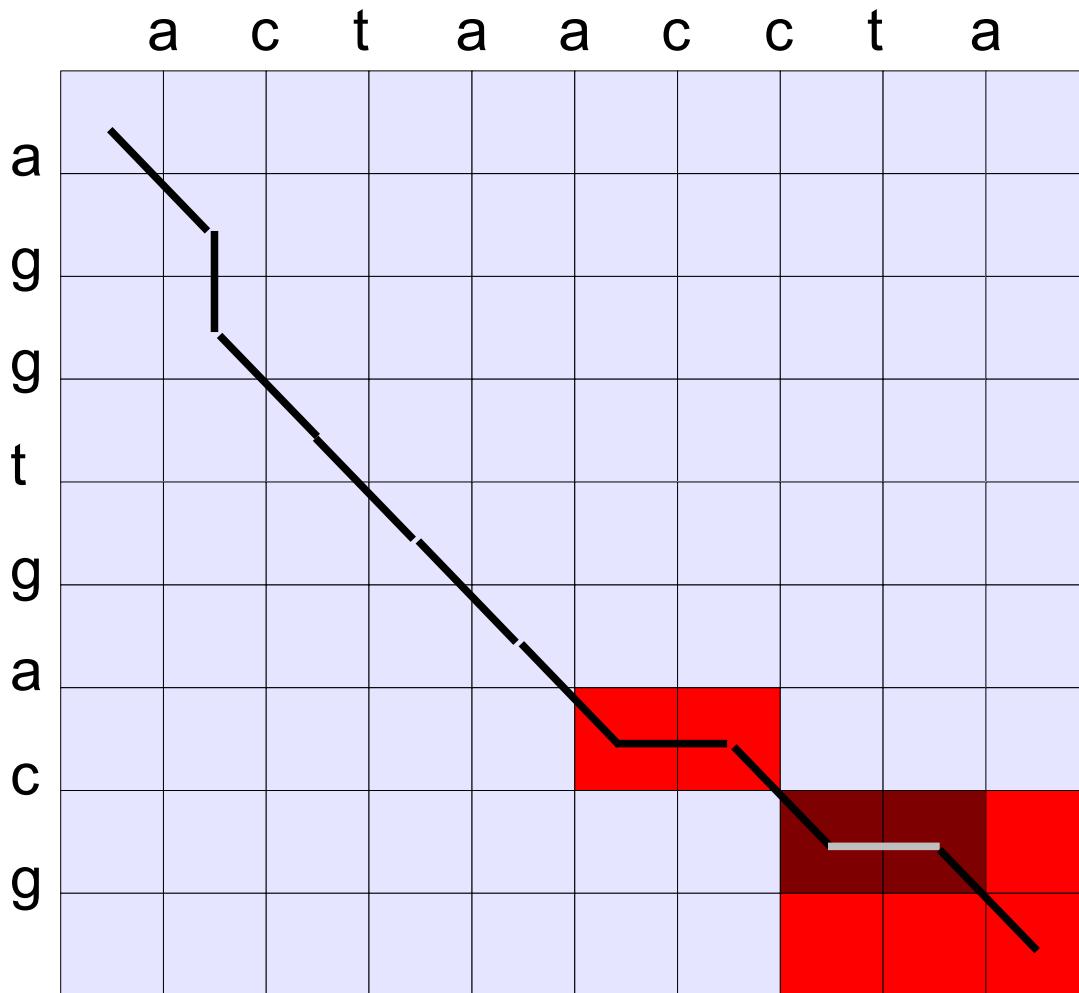
Hirschberg's idea:

Find “middle edge” (i.e. “middle column”) of an optimal alignment, split into two subproblems and recurse ...

Time:

$nm +$
 $(n / 2)m +$
 $(n / 4)m +$
 $(n / 8)m$

Linear space – Hirschberg's idea



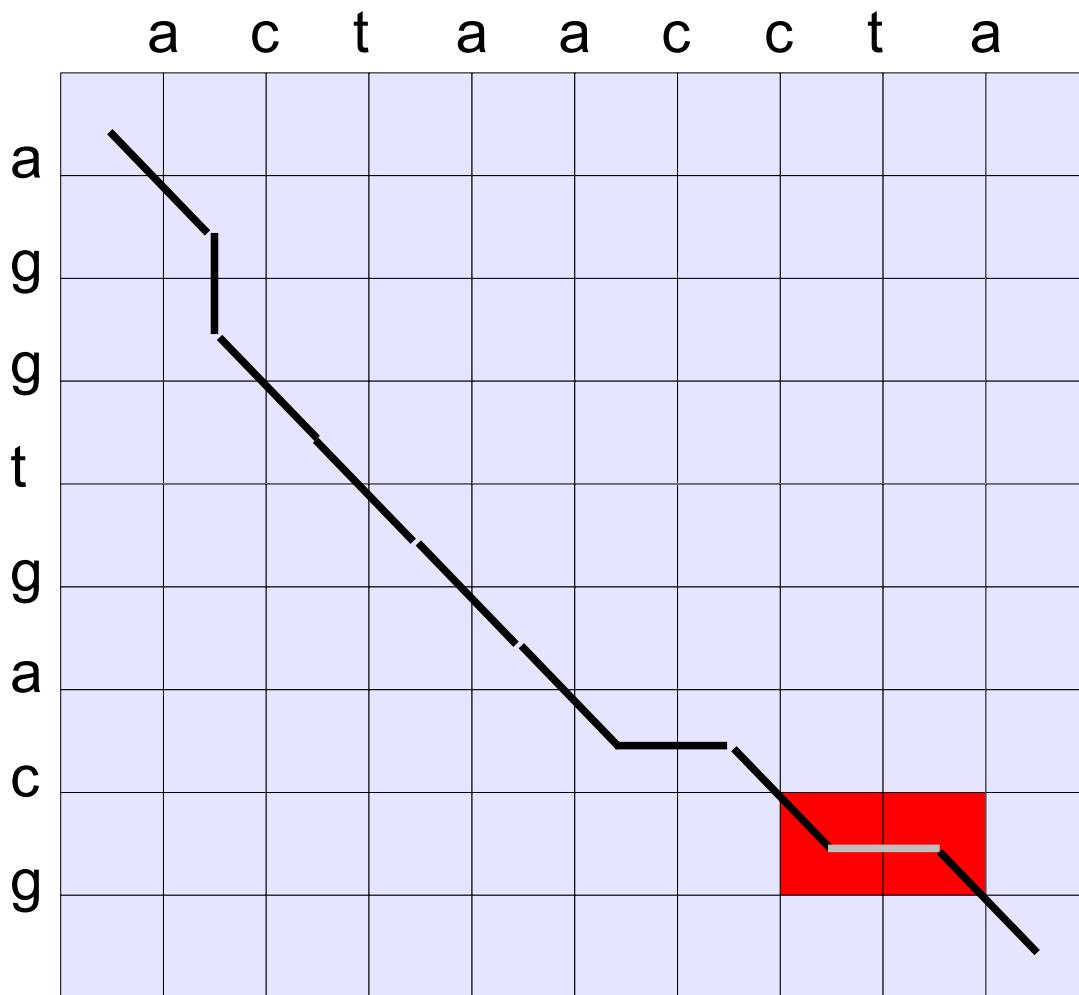
Hirschberg's idea:

Find “middle edge” (i.e. “middle column”) of an optimal alignment, split into two subproblems and recurse ...

Time:

$nm +$
 $(n / 2)m +$
 $(n / 4)m +$
 $(n / 8)m$

Linear space – Hirschberg's idea



Hirschberg's idea:

Find “middle edge” (i.e. “middle column”) of an optimal alignment, split into two subproblems and recurse ...

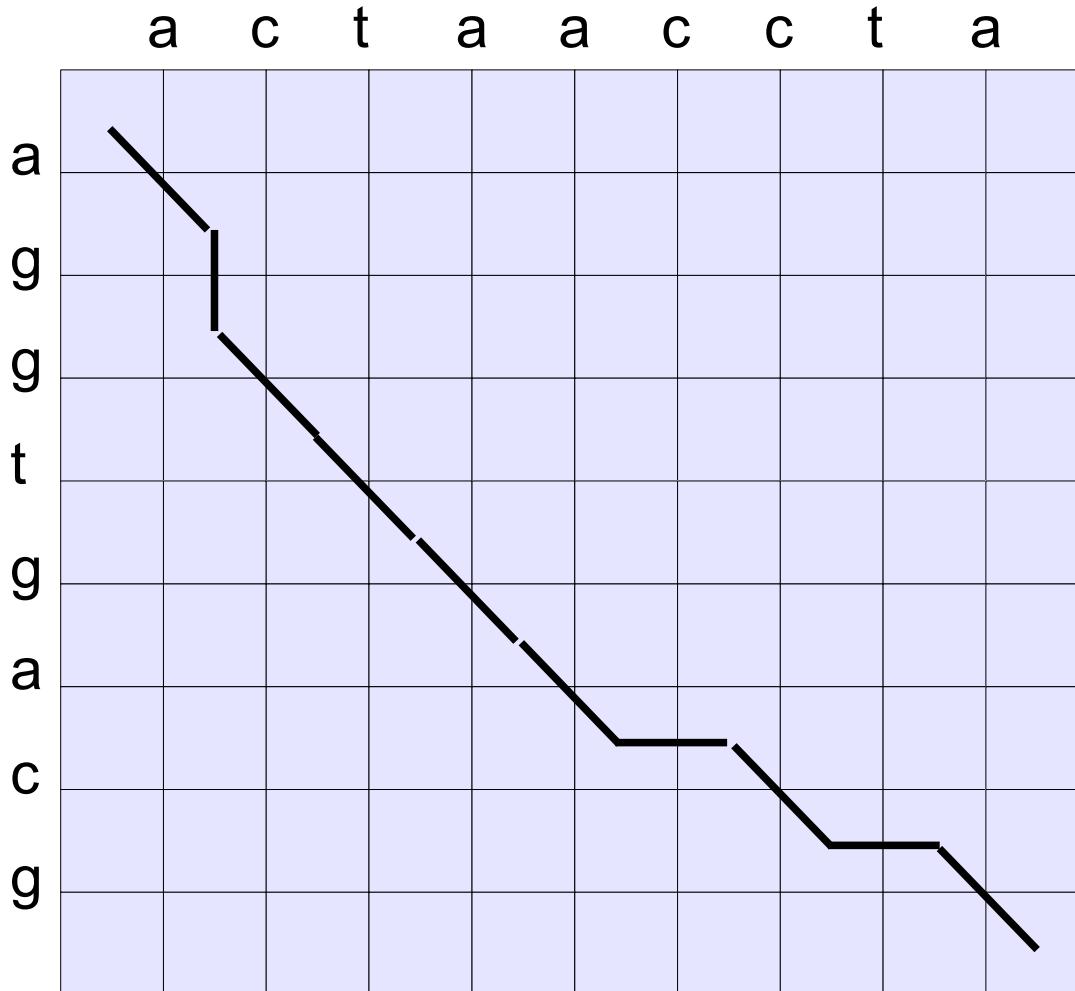
Time:

$nm +$
 $(n / 2)m +$
 $(n / 4)m +$
 $(n / 8)m$

Time analysis (in general):

$$nm + (n/2)m + (n/4)m + \dots + 2m = m(n + (n/2) + (n/4) + \dots + 2) = \Theta(mn)$$

because $n \leq n + (n/2) + (n/4) + \dots + 2 \leq 2n$



Hirschberg's idea:

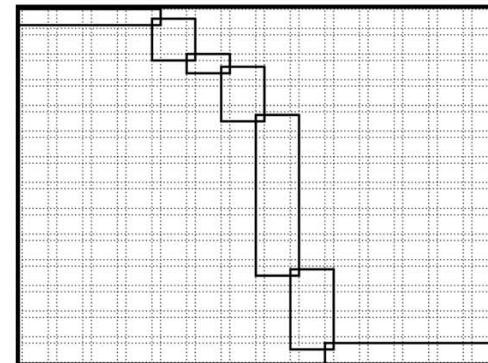
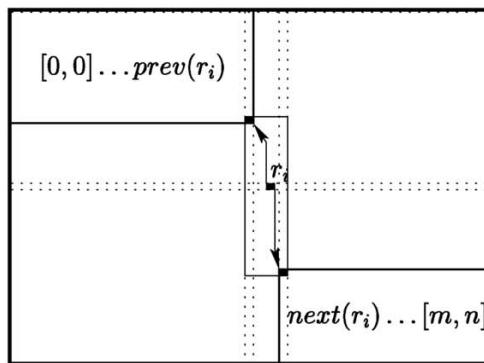
Find “middle edge” (i.e. “middle column”) of an optimal alignment, split into two subproblems and recurse ...

Time:

$$\begin{aligned} & nm + \\ & (n / 2)m + \\ & (n / 4)m + \\ & (n / 8)m \end{aligned}$$

Parallel sequence alignment

- For **coarse-grained parallelism**, exploit Hirschberg's recursion
- Naïve application of the recursion by itself is suboptimal
- Better variants are known that are extensions over Hirschberg



Result: The sequence alignment problem can be solved in **$O((m+n)/p)$ space** and **$O(mn/p)$ time**

Rajko S, Aluru S. Space and time optimal parallel sequence alignments. IEEE Transactions on Parallel and Distributed Systems. 2004 Dec;15(12):1070-81.

Outline

- Genome (and metagenome) assembly problems
- Indexing with hash tables
- Pairwise sequence alignment
- **Indexing with suffix arrays**

Suffix Arrays

Suffix arrays are alternatives to hash tables for string indexing and compression.

Suffix array SA of x is an array of n integers such that:

$SA[i] = j$ implies that suffix j of x has rank i in the lexicographical ordering of all suffixes of x

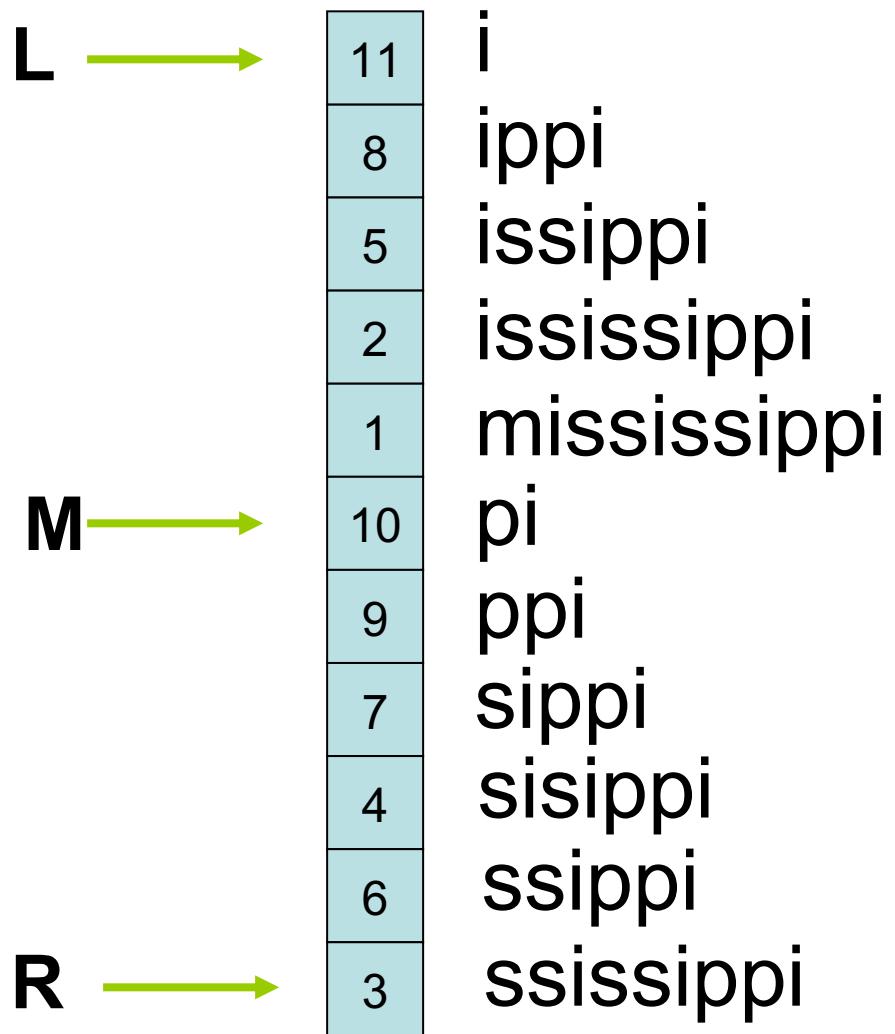
SA augmented with **Longest Common Prefix (LCP)** array matches the bounds and functionality of suffix trees. One can construct SAs naively in $O(n^2 \log(n))$ time using string sorting. **We can do much better!**

Suffix Array Example

Let $S = \text{mississippi}$

Let $P = \text{issa}$

Main motivation:
every possible
substring is a
prefix of a suffix



Linear time construction

Kärkkäinen and Sanders's idea for suffix array construction

Step 1

Construct the suffix array of the suffixes starting at positions $i \bmod 3 \neq 0$.

... done recursively by reducing the problem to a string of 2/3 size ...

Step 2

Construct the suffix array of the remaining suffixes.

... done using the suffix array constructed in step 1 ...

Step 3

Merge the two suffix arrays into one

Step 1 - Compute SA_{12}

Suffixes $i \bmod 3 \neq 0$

- 1: ississippi
- 2: ssissippi
- 4: issippi
- 5: ssippi
- 7: ippi
- 8: ppi
- 10: i

m i s s i s s i p p i
0 1 2 3 4 5 6 7 8 9 10

Step 1 - Compute SA_{12}

Suffixes $i \bmod 3 \neq 0$

- 1: ississippi
- 2: ssissippi
- 4: issippi
- 5: ssiippi
- 7: ipp*i*
- 8: ppi
- 10: i\$\$

↑
sentinels

Sort by prefix in time $O(n)$

- 10: i\$\$
- 7: ipp
- 1: iss
- 4: iss
- 8: ppi
- 2: ssi
- 5: ssi

m i s s i s s i p p i
0 1 2 3 4 5 6 7 8 9 10

Step 1 - Compute SA_{12}

Suffixes $i \bmod 3 \neq 0$

- 1: ississippi
- 2: ssissippi
- 4: issippi
- 5: ssiippi
- 7: ippi
- 8: ppi
- 10: i\$\$

Sort by prefix in time $O(n)$

- 10: i\$\$ 1
- 7: ipp 2
- 1: iss 3
- 4: iss 3
- 8: ppi 4
- 2: ssi 5
- 5: ssi 5

lexicographical
naming

If no suffix is assigned the same lex-name, we are done

m i s s i s s i p p i
0 1 2 3 4 5 6 7 8 9 10

Step 1 - Compute SA^{12}

Suffixes $i \bmod 3 \neq 0$

- 1: ississippi
- 2: ssissippi
- 4: issippi
- 5: ssiippi
- 7: ippi
- 8: ppi
- 10: i\$\$

Sort by prefix in time $O(n)$

- 10: i\$\$ 1
- 7: ipp 2
- 1: iss 3
- 4: iss 3
- 8: ppi 4
- 2: ssi 5
- 5: ssi 5

lexicographical
naming

If no suffix is assigned the same lex-name, we are done, otherwise:

$u = \text{"lex-names for } i \bmod 3 = 1\text{"} \# \text{"lex-names for } i \bmod 3 = 2\text{"} = 3 3 2 1 \# 5 5 4$

where # is a special character not occurring anywhere else. The suffix array $SA(u)$ of u implies SA^{12} as there is a 1-1 mapping from suffixes of u and suffixes of $s[i..n]$ where $i \bmod 3 \neq 0$

Step 1 - Compute SA^{12}

Suffixes $i \bmod 3 \neq 0$

1: **i**s*s*issippi
2: **s**i*s*issippi
4: **i****s**issippi
5: **s****i**ppi
7: i**p**pi
8: p**p**i
10: i\$b\$

Sort by prefix in time $O(n)$

10: i\$b\$ 1
7: ip**p** 2
1: i**s**s 3
4: i**s**s 3
8: p**p**i 4
2: \$ 5

Time: $T(n) = O(n) + T(2n/3) = O(n)$

If no suffix is assigned the same lex-name, we are done, otherwise:

$u = \text{"lex-names for } i \bmod 3 = 1\text{"} \# \text{"lex-names for } i \bmod 3 = 2\text{"} = 3\ 3\ 2\ 1\ \#\ 5\ 5\ 4$

where # is a special character not occurring anywhere else. The suffix array $SA(u)$ of u implies SA^{12} as there is a 1-1 mapping from suffixes of u and suffixes of $s[i..n]$ where $i \bmod 3 \neq 0$

Step 1 - Why ...

$$u = \begin{matrix} 3 & 3 & 2 & 1 & \# & 5 & 5 & 4 \\ iss & iss & ipp & i\$ & & ssi & ssi & ppi \end{matrix}$$

Lexicographical names

10:	i\$\$	1
7:	ipp	2
1:	iss	3
4:	iss	3
8:	ppi	4
2:	ssi	5
5:	ssi	5

Step 1 - Why ...

$$u = 3 \ 3 \ 2 \ 1 \ # \ 5 \ 5 \ 4$$

iss iss ipp i\$\$ ssi ssi ppi

Suffixes of u

- 0: 3321#554
- 1: 321#554
- 2: 21#554
- 3: 1#554
- 4: 554
- 5: 54
- 6: 4

Step 1 - Why ...

$$u = \begin{matrix} 3 & 3 & 2 & 1 & \# & 5 & 5 & 4 \\ \text{iss} & \text{iss} & \text{ipp} & \text{i\$\$} & \text{ssi} & \text{ssi} & \text{ppi} \end{matrix}$$

Suffixes of u

0:	3321#554	ississippi\$\$#ssissippi
1:	321#554	issippi\$\$#ssissippi
2:	21#554	ippi\$\$#ssissippi
3:	1#554	i\$\$#ssissippi
4:	554	ssissippi
5:	54	ssippi
6:	4	ppi

Lex-names expanded to corresponding prefixes of length 3

Step 1 - Why ...

$$\mathbf{u} = \begin{matrix} 3 & 3 & 2 & 1 & \# & 5 & 5 & 4 \\ \text{iss} & \text{iss} & \text{ipp} & \text{i\$\$} & \text{ssi} & \text{ssi} & \text{ppi} \end{matrix}$$

Suffixes of \mathbf{u}

0:	3321#554	ississippi\$\$#ssissippi
1:	321#554	issippi\$\$#ssissippi
2:	21#554	ippi\$\$#ssissippi
3:	1#554	i\$\$#ssissippi
4:	554	ssissippi
5:	54	ssippi
6:	4	ppi

Since the special character # is unique and occurs in unique positions,
we can sort these strings by sorting the prefixes (of \mathbf{u}) up to #

Step 1 - Why ...

$$u = 3 \ 3 \ 2 \ 1 \ # \ 5 \ 5 \ 4$$

iss iss ipp i\$\$ ssi ssi ppi

$i < j$ iff $xxx < yyy$, i.e $1 < 3$ because $i\$ \$ < i\ s\ s$

Suffixes of u

0:	3321#554	ississippi\$\$#ssissippi
1:	321#554	issippi\$\$#ssissippi
2:	21#554	ippi\$\$#ssissippi
3:	1#554	i\$\$#ssissippi
4:	554	ssissippi
5:	54	ssi
6:	4	ppi

Lex-names expanded to corresponding prefixes of length 3

```
1: ississippi  
2: ssissippi  
4: issippi  
5: ssippi  
7: ippi  
8: ppi  
10: i$$
```

Step 1 - Why ...

$$u = 3 \ 3 \ 2 \ 1 \ # \ 5 \ 5 \ 4$$

iss iss ipp i\$\$ ssi ssi ppi

Suffixes of u

0:	3321#554	ississippi\$\$#ssissippi	1
1:	321#554	issippi\$\$#ssissippi	4
2:	21#554	ippi\$\$#ssissippi	7
3:	1#554	i\$\$#ssissippi	10
4:	554	ssissippi	2
5:	54	ssippi	5
6:	4	ppi	8

Since the sentinel \$ is unique and occurs in unique positions, we can sort these strings by sorting the prefixes (of u) up to the sentinel \$...

```

1: ississippi
2: ssissippi
4: issippi
5: ssippi
7: ippi
8: ppi
10: i$$

```

Step 1 - Why ...

$$u = 3 \ 3 \ 2 \ 1 \ # \ 5 \ 5 \ 4$$

iss iss ipp i\$\$ ssi ssi ppi

Suffixes of u

0:	3321#554	ississippi\$\$#ssissippi	1
1:	321#554	issippi\$\$#ssissippi	4
2:	21#554	ippi\$\$#ssissippi	7
3:	1#554	i \$\$#ssissippi	10
4:	554	ssissippi	2
5:	54	ssippi	5
6:	4	ppi	8

Suffix array

3
2
1
0
6
5
4

Since the sentinel \$ is unique and occurs in unique positions, we can sort these strings by sorting the prefixes (of u) up to the sentinel \$...

```

1: ississippi
2: ssissippi
4: issippi
5: ssippi
7: ippi
8: ppi
10: i$$

```

Step 1 - Why ...

$$u = 3 \ 3 \ 2 \ 1 \ # \ 5 \ 5 \ 4$$

iss iss ipp i\$\$ ssi ssi ppi

SA^{12}

Suffixes of u

0:	3321#554	ississippi\$\$#ssissippi	1
1:	321#554	issippi\$\$#ssissippi	4
2:	21#554	ippi\$\$#ssissippi	7
3:	1#554	i\$\$#ssissippi	10
4:	554	ssissippi	2
5:	54	ssippi	5
6:	4	ppi	8

Suffix array

3	10
2	7
1	4
0	1
6	8
5	5
4	2

Since the sentinel \$ is unique and occurs in unique positions, we can sort these strings by sorting the prefixes (of u) up to the sentinel \$...

Step 2 - Compute SA^0

Suffixes $i \bmod 3 = 0$

0: mississippi

3: sissippi

6: sippi

9: pi

m i s s i s s i p p i
0 1 2 3 4 5 6 7 8 9 10

Step 2 - Compute SA^0

Suffixes $i \bmod 3 = 0$

- $0:$ mississippi $m s[1..]$
- $3:$ sissippi $s s[4..]$
- $6:$ sippi $s s[7..]$
- $9:$ pi $p s[10..]$

Idea: every suffix $i \bmod 3 = 0$ can be written $s[i] s[i+1..]$ where $i+1 \bmod 3 \neq 0$, i.e. the ordering of $s[i+1..]$ is known c.f. $SA^{12} \dots$

10	7	4	1	8	5	2
----	---	---	---	---	---	---

$m \quad i \quad s \quad s \quad i \quad s \quad s \quad i \quad p \quad p \quad i$
 $0 \quad 1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \quad 7 \quad 8 \quad 9 \quad 10$

Step 2 - Compute SA^0

Suffixes $i \bmod 3 = 0$

- 0: mississippi m s[1..]
- 3: sissippi s s[4..]
- 6: sippi s s[7..]
- 9: pi p s[10..]

Sort by first letter and
use (inverse) SA^{12} to
solve ties:

- 0: m s[1..]
- 9: p s[10..]
- 6: s s[7..]
- 3: s s[4..]

The order of suffix 6 and 3 is by the position of $s[7..]$ and $s[4..]$ in SA^{12}
e w [i+1..] is know c.f. $SA^{12} \dots$

10	7	4	1	8	5	2
----	---	---	---	---	---	---

m i s s i s s i p p i
0 1 2 3 4 5 6 7 8 9 10

Step 2 - Compute SA^0

Suffixes $i \bmod 3 = 0$

- 0: mississippi m s[1..]
- 3: sissippi s s[4..]
- 6: sippi s s[7..]
- 9: pi p s[10..]

Time: $O(n)$

Sort by first letter and
use (inverse) SA^{12} to
solve ties:

- 0: m s[1..]
- 9: p s[10..]
- 6: s s[7..]
- 3: s s[4..]

The order of suffix 6 and 3 is by the
position of s[7..] and s[4..] in SA^{12}



10	7	4	1	8	5	2
----	---	---	---	---	---	---

m i s s i s s i p p i
0 1 2 3 4 5 6 7 8 9 10

Step 3 - Merging

Idea: every suffix j can be written $s[j] s[j+1..]$ and $s[j] s[j+1] s[j+2..]$, where $j+1 \bmod 3 \neq 0$ and/or $j+2 \bmod 3 \neq 0$, and the ordering of suffixes $s[i..]$ for $i \bmod 3 \neq 0$ is known c.f. $SA^{12\dots}$

0	9	6	3
---	---	---	---

10	7	4	1	8	5	2
----	---	---	---	---	---	---

Example

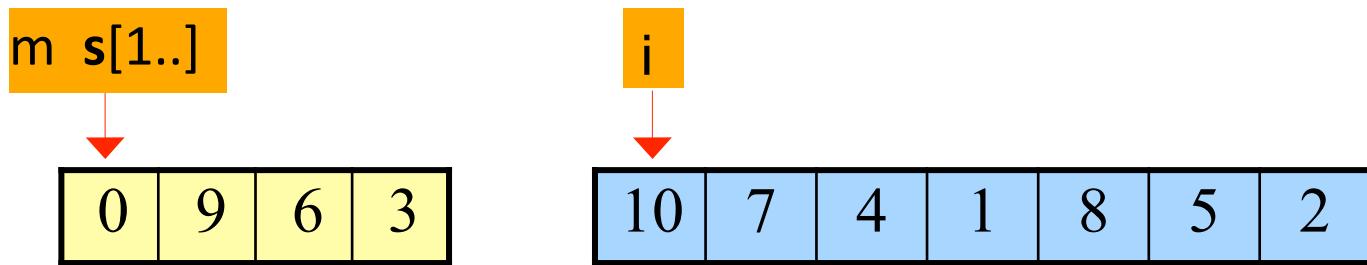
Determine order of suffix $s[1..]$ and $s[4..]$

$$s[1..] = i s[2..]$$

$$s[4..] = i s[5..]$$

$$s[5..] < s[2..], \text{ i.e. } s[4..] < s[1..]$$

Step 3 - Merging



SA:

m i s s i s s i p p i
0 1 2 3 4 5 6 7 8 9 10

We will not go into it here but merging can be done in **$O(n)$** time

Time (DC3): $T(n) = O(n) + T(2n/3) = O(n)$

Skew Algorithm in Parallel (pDC3)

Parallel sort
Prefix sum
AlltoAllV
AlltoAllV
Data parallel splitting

Function $pDC3(T)$

$S := \langle ((T[i, i+2]), i) : i \in [0, n), i \bmod 3 \neq 0 \rangle$

sort S by the first component

$P := name(S)$

if the names in P are not unique then

permute the $(r, i) \in P$ such that they are sorted by $(i \bmod 3, i \bmod 3)$

$SA^{12} := pDC3(\langle c : (c, i) \in P \rangle)$

$P := \langle (j + 1, SA^{12}[j]) : j \in [0, 2n/3] \rangle$

permute P such that it is sorted by the second component

$S_0 := \langle (T[i], T[i+1], c', c'', i) : i \bmod 3 = 0, (c', i+1), (c'', i+2) \in P \rangle$

$S_1 := \langle (c, T[i], c', i) : i \bmod 3 = 1, (c, i), (c', i+1) \in P \rangle$

$S_2 := \langle (c, T[i], T[i+1], c'', i) : i \bmod 3 = 2, (c, i), (c'', i+2) \in P \rangle$

$S := \text{sort } S_0 \cup S_1 \cup S_2 \text{ using comparison function:}$

$(c, \dots) \in S_1 \cup S_2 \leq (d, \dots) \in S_1 \cup S_2 \Leftrightarrow c \leq d$

$(t, t', c', c'', i) \in S_0 \leq (u, u', d', d'', j) \in S_0 \Leftrightarrow (t, c') \leq (u, d')$

$(t, t', c', c'', i) \in S_0 \leq (d, u, d', j) \in S_1 \Leftrightarrow (t, c') \leq (u, d')$

$(t, t', c', c'', i) \in S_0 \leq (d, u, u', d'', j) \in S_2 \Leftrightarrow (t, t', c'') \leq (u, u', d'')$

return $\langle \text{last component of } s : s \in S \rangle$

pDC3 Complexity

$$T_{pDC3} = T_{\text{par sort}}(n, p) + T_{\text{all-to-all}}(n/p, p) + f(p) \log(n)$$

Bounds hold as long as

$$T_{\text{par sort}}(2n/3, p) \leq 2/3 T_{\text{par sort}}(n, p) + f(p)$$

same for T_{all2all}

$f(p) \in \Omega(\log(p))$ is an implementation dependent bottleneck term

Fabian Kulla and Peter Sanders. "Scalable parallel suffix array construction." Parallel Computing 33.9 (2007): 605-612