CS 267
Lecture 26: Computational Biology

Aydin Buluc

Slide credits:

Evangelos Georganas, Giulia Guidi, Carl Kingsford, Oguz Selvitopi, Christian Nørgaard Storm Pedersen, Kathy Yelick

https://sites.google.com/lbl.gov/cs267-spr2021/

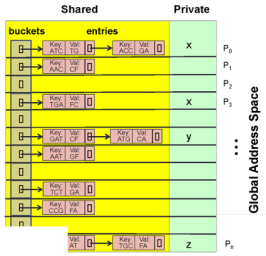
Outline

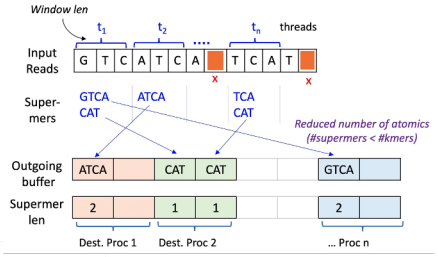
Computational Biology is a huge field; some will claim it is not even a coherent field. It has both simulation (e.g. cell simulations) and data analysis (e.g. –omics) aspects.

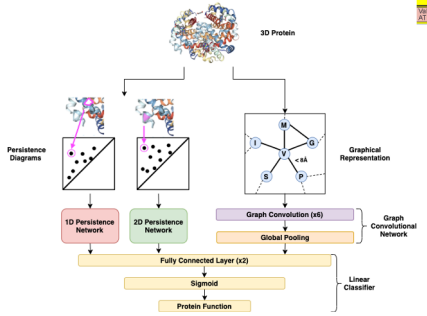
This lecture focuses on -omics aspects:

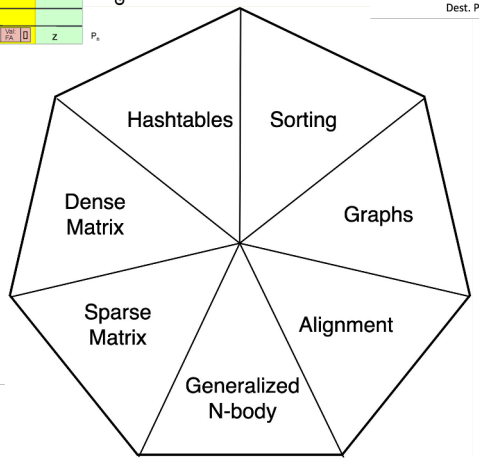
- Genome (and metagenome) assembly problems
- Indexing with hash tables
- The sparse matrix approach
- Protein Family Identification
- Pairwise sequence alignment
- Indexing with suffix arrays

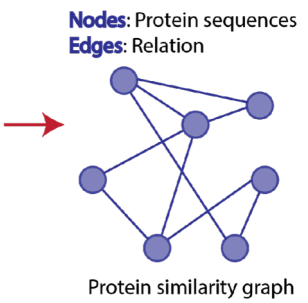
Motifs of Genomic Data Analysis

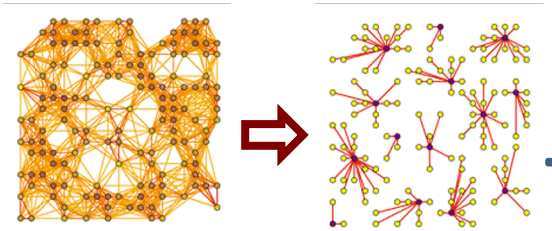


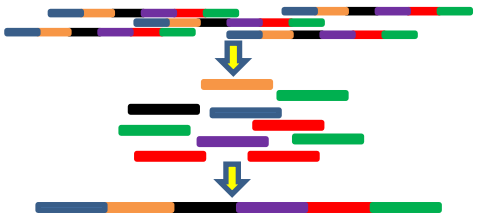


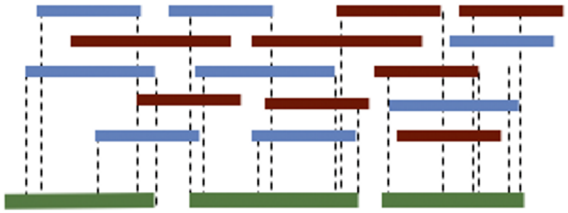












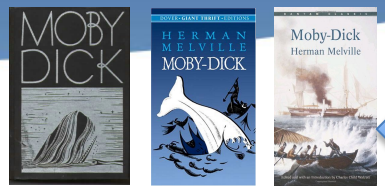
Yelick, et al. “The Parallelism Motifs of Genomic Data Analysis”, Philosophical Transactions A, 2020

Outline

- **Genome (and metagenome) assembly problems**
- Indexing with hash tables
- The sparse matrix approach
- Protein Family Identification
- 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

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
great are made so
all men tragically g
ally great
```

5. Reconstruct original DNA sequence from the read set.

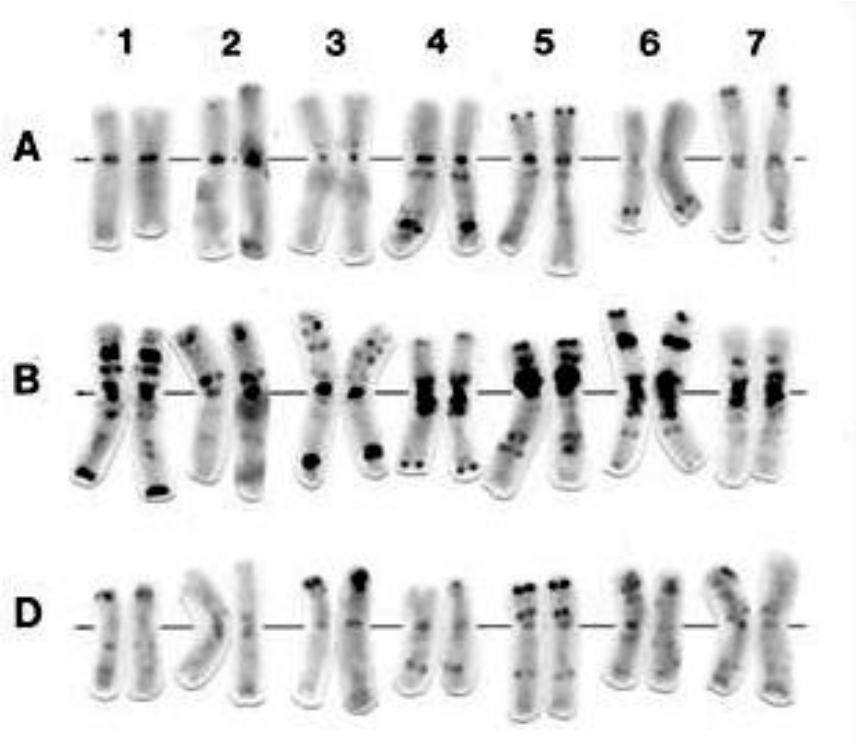
```
ACCGTAGCAAAACCGGGTAGTCATACTACTACGTACTCATCT
ACCGTAGCAA
GTAGTCATACT
AAACCGGGTA
CTACTACGTAC
CGTACTCATCT
```

De novo Genome Assembly is hard

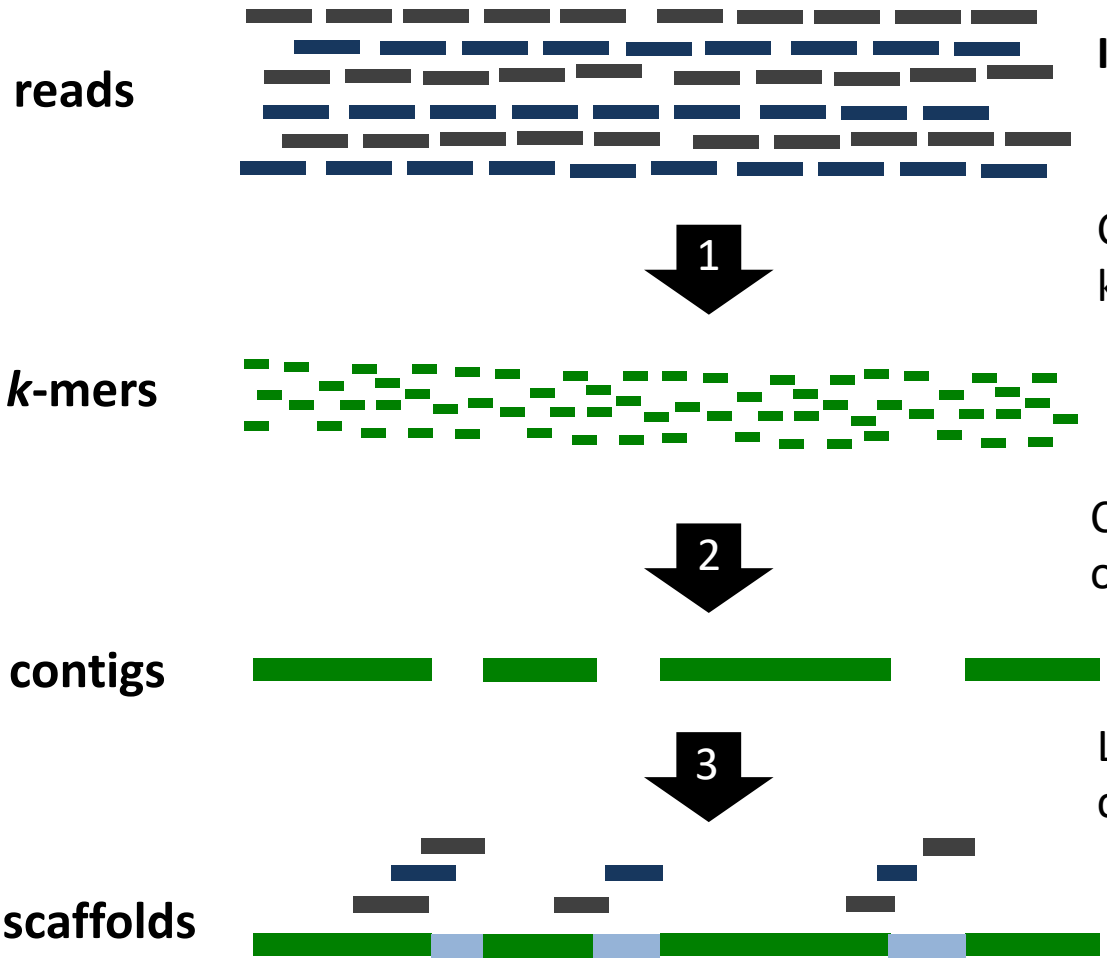
- There is no genome reference!
  - In principle we want to reconstruct unknown genome sequence.
- Reads are significantly shorter than whole genome.
  - Reads consist of 20 to 30K bases
  - Genomes vary in length and complexity – up to 30G bases
- Reads include **errors**.
- Genomes have repetitive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



Input: Reads that may contain errors

Chop reads into k-mers, process k-mers to **exclude errors**

I/O, bandwidth, and memory intensive

Construct & traverse de Bruijn graph of k-mers, generate contigs

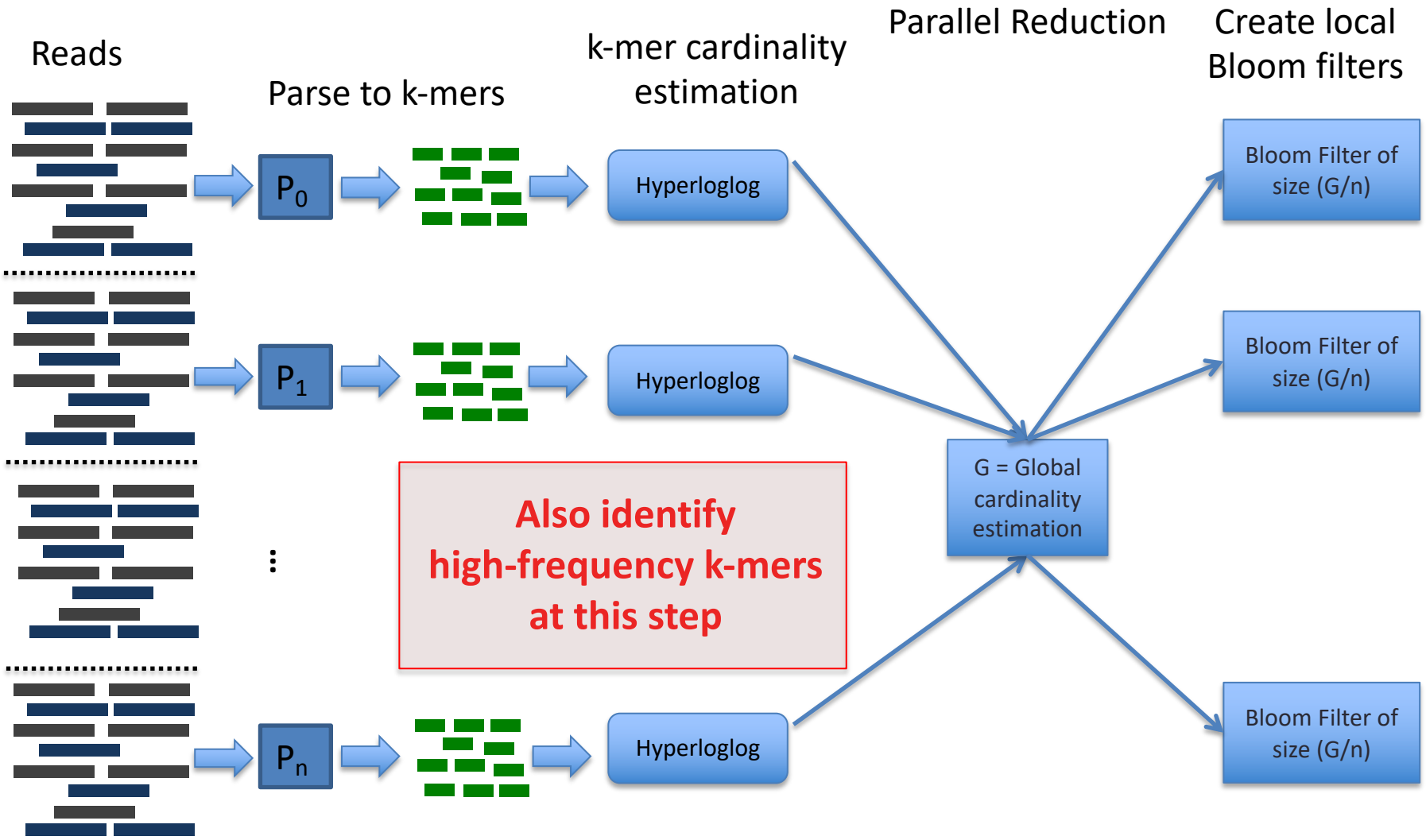
Latency bound (irregular accesses)

Leverage read information to link contigs and generate scaffolds.

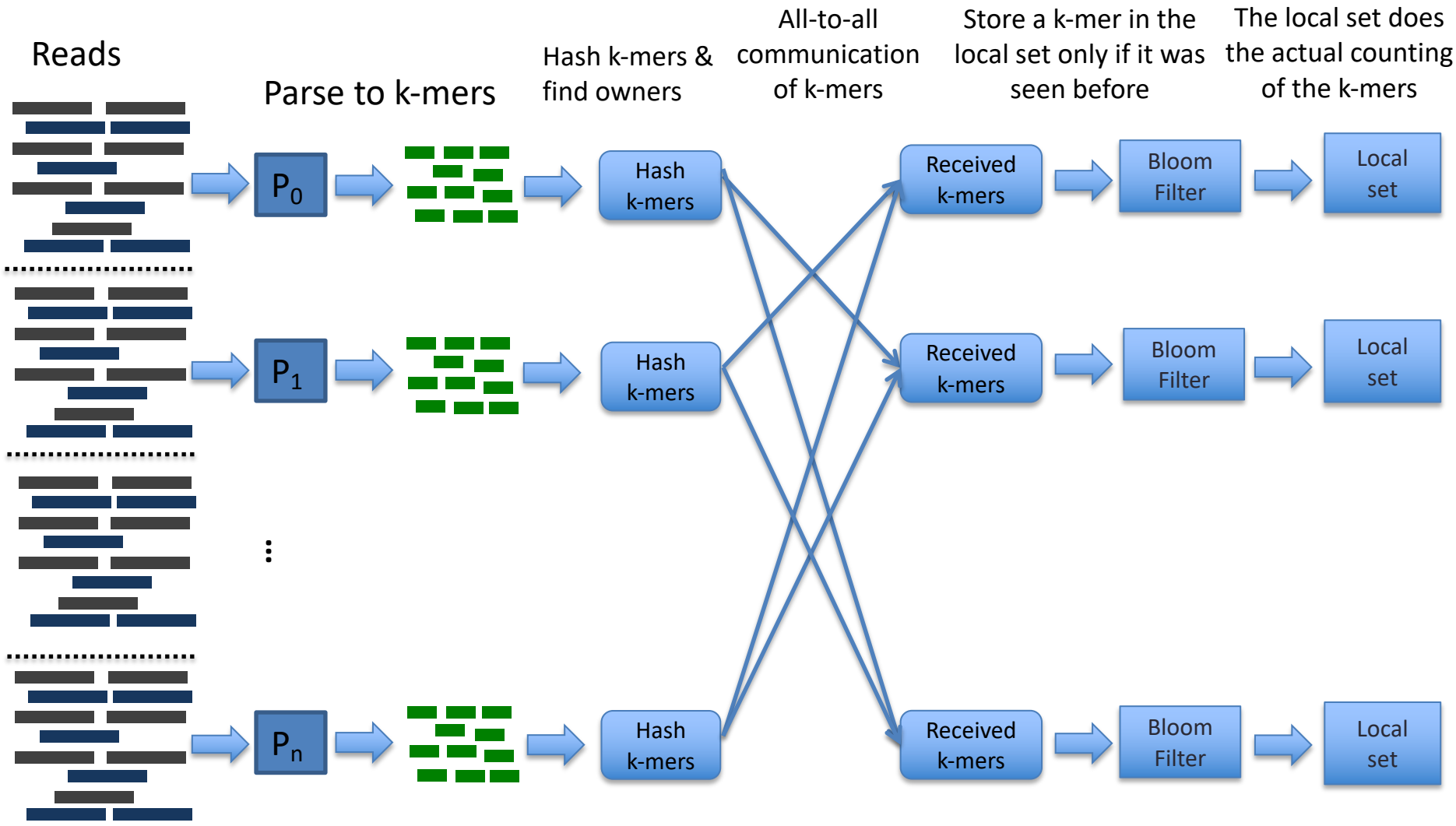
Compute and I/O intensive

Georganas, E., Buluç, A., Chapman, J., Hofmeyr, S., Aluru, C., Egan, R., Oliker, L., Rokhsar, D. and Yelick, 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?

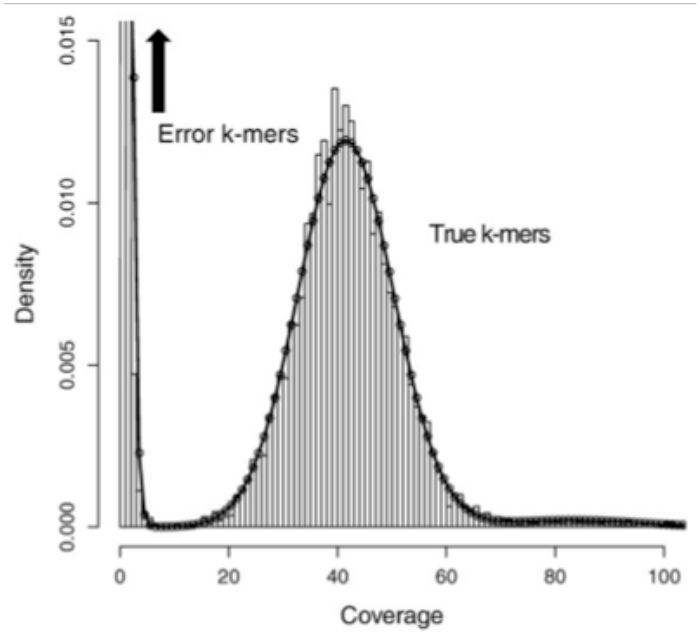
Bloom filter is a *probabilistic* data structure used for membership queries

- Given a bloom filter, we can ask:

"Have we seen this k-mer before?"

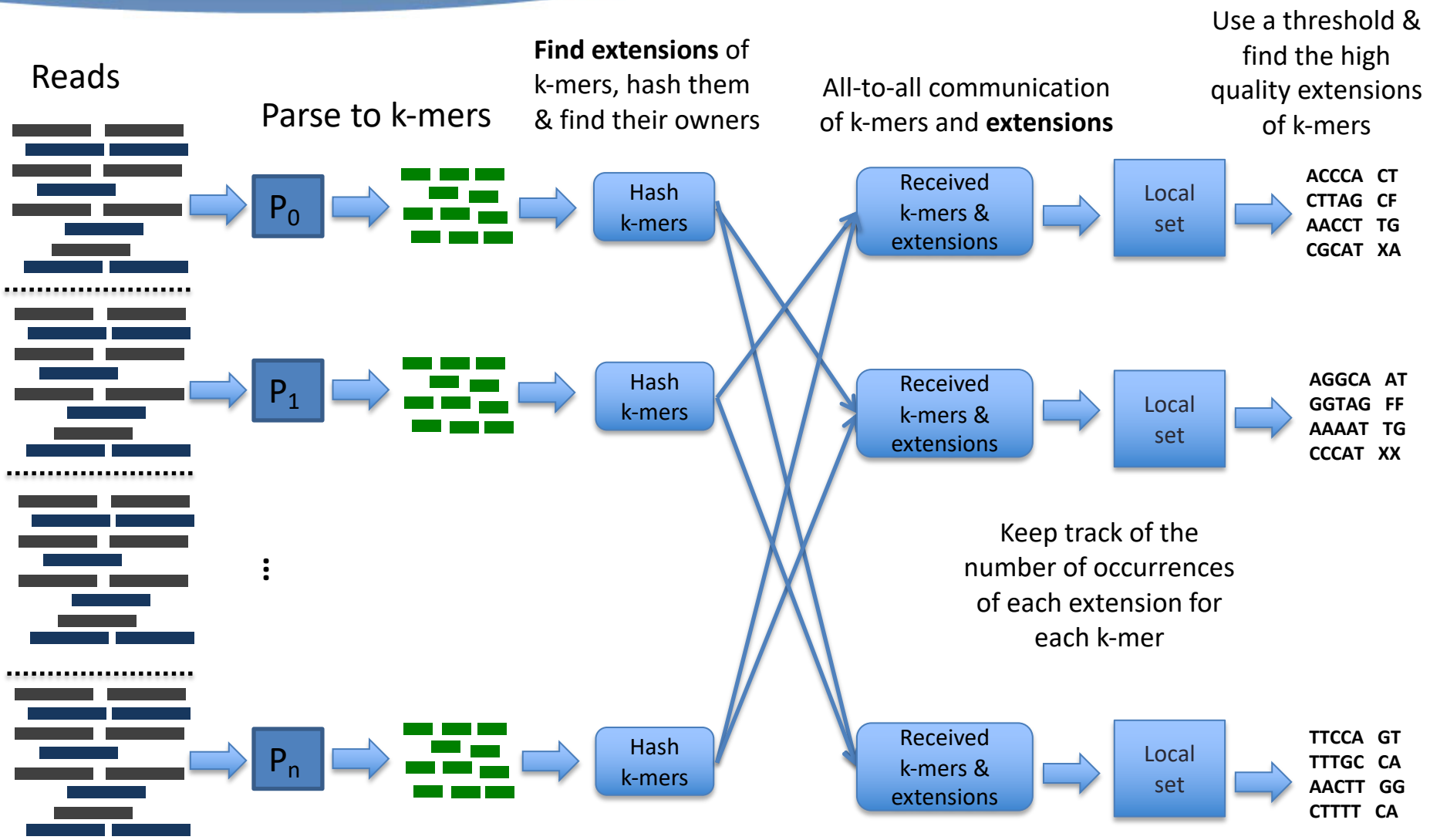
- **No false negatives.**
- May have false positives

(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

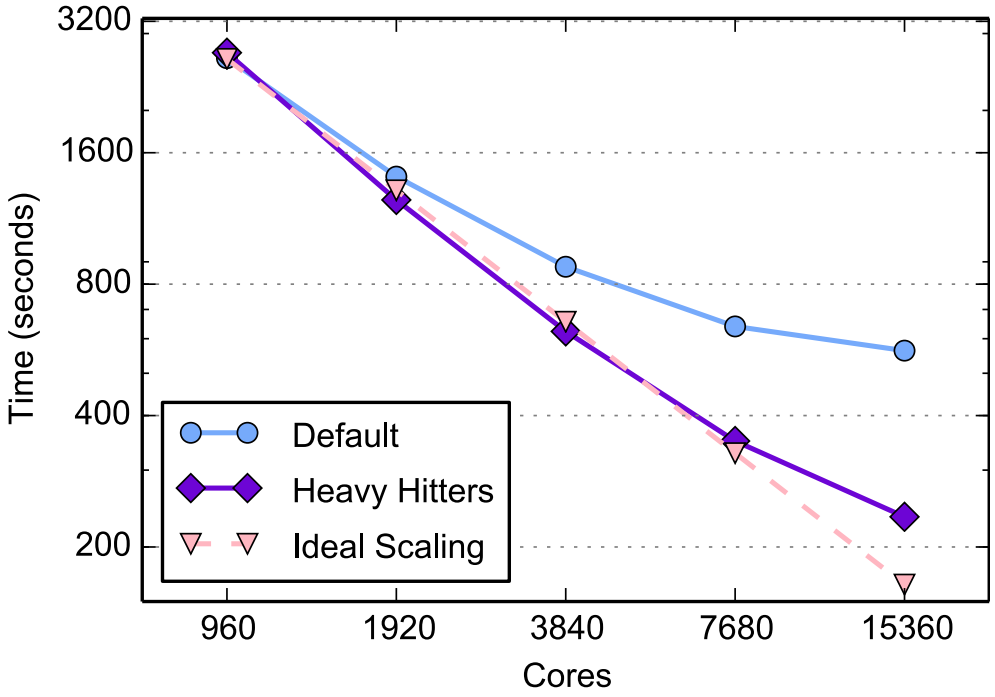


High-frequency k-mers

Long-tailed distribution for genomes with repetitive content:

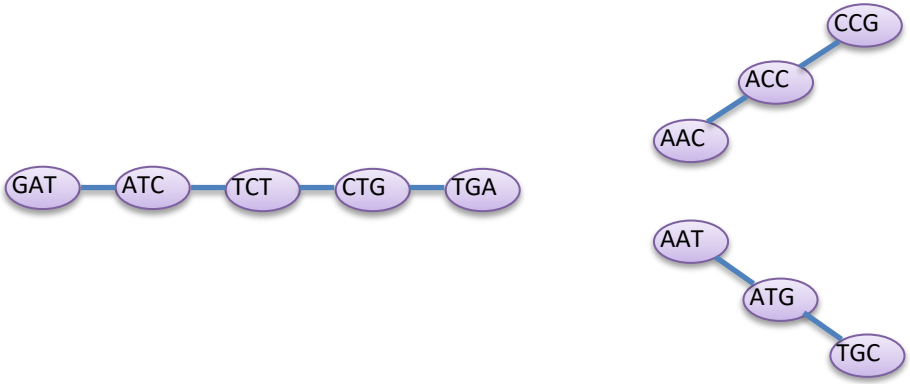
- The maximum count for any k-mer in the wheat dataset is **451 million**
- Our original scheme (SC'14) was "owner counts", after an all-to-all
- Counting an item w/ 451 million occurrences alone is **load imbalanced**

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



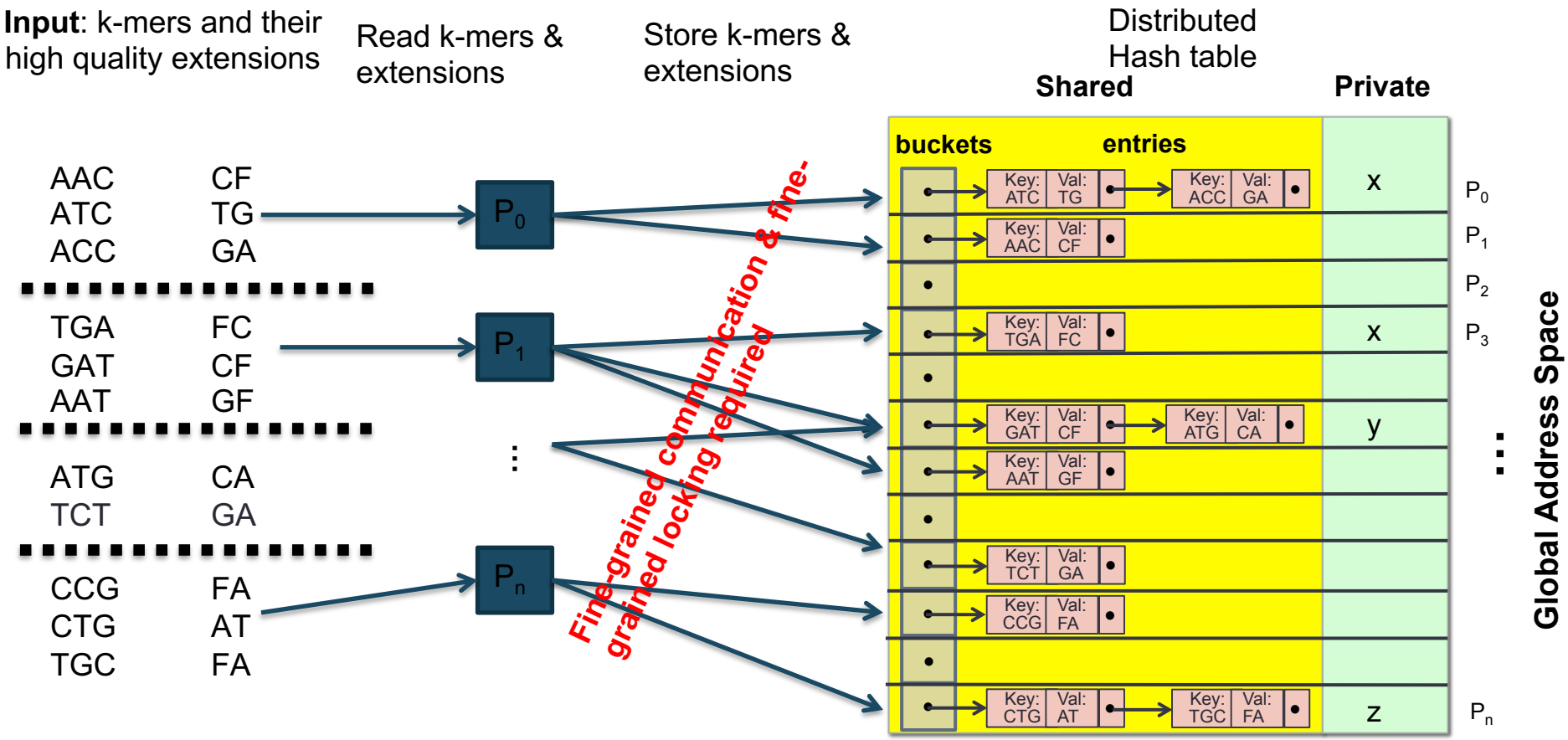
Parallel De Bruijn Graph Construction

- 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



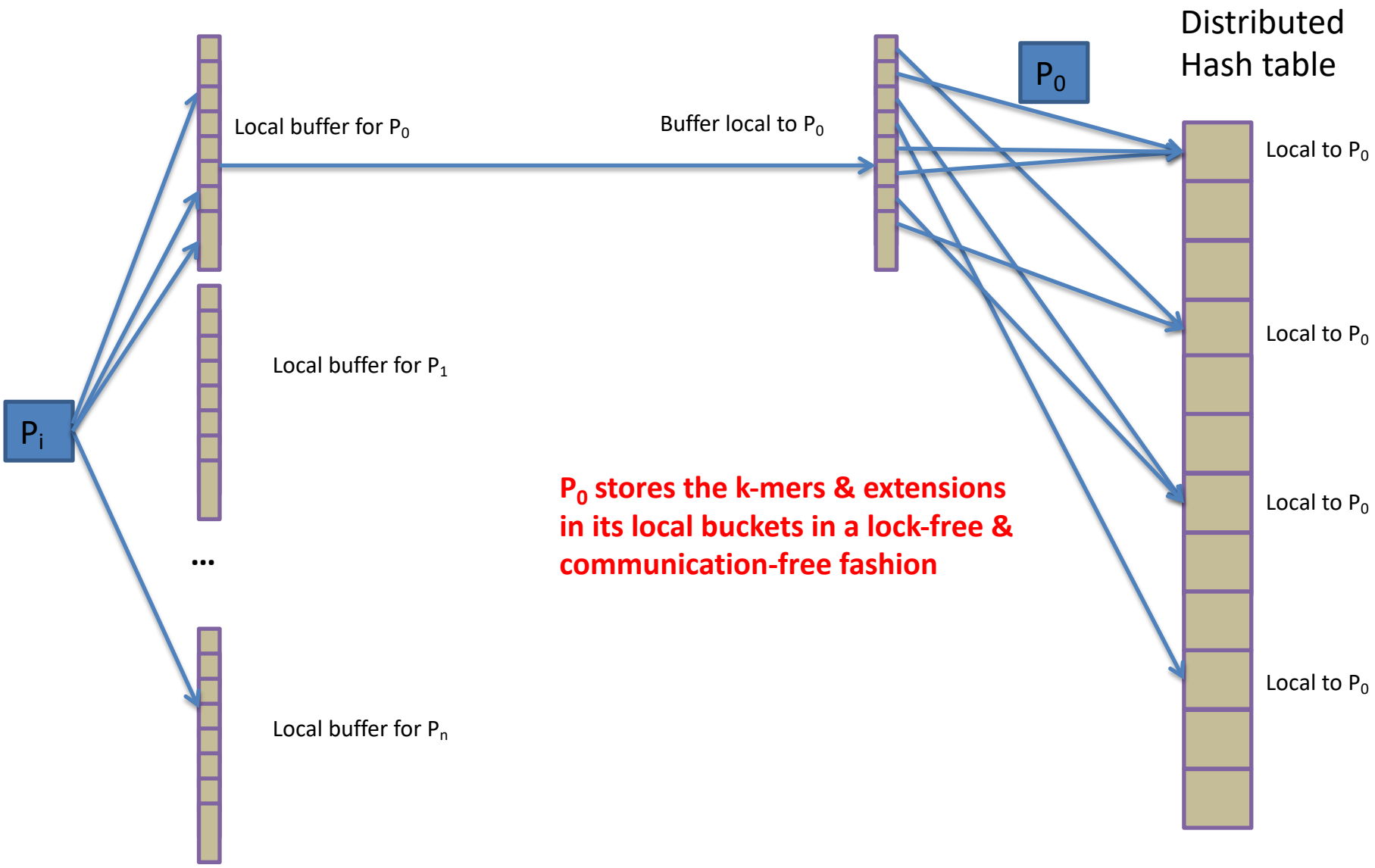
Parallel Graph Construction Challenge

- **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.**

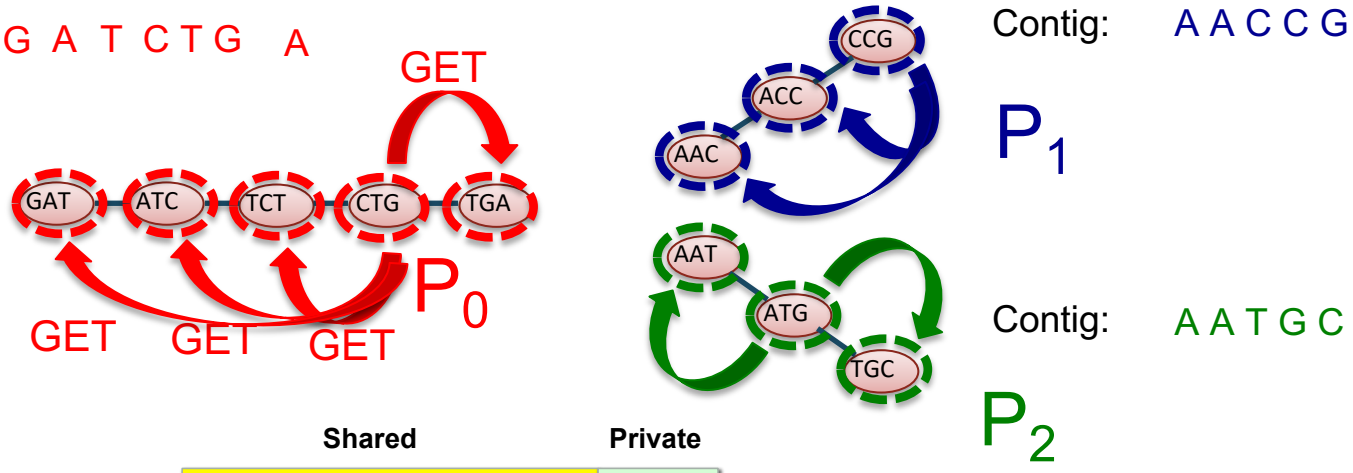
Georganas, E., Buluç, A., Chapman, J., Oliker, L., Rokhsar, D. and Yelick, K. Parallel de bruijn graph construction and traversal for de novo genome assembly. In *SC'14*.

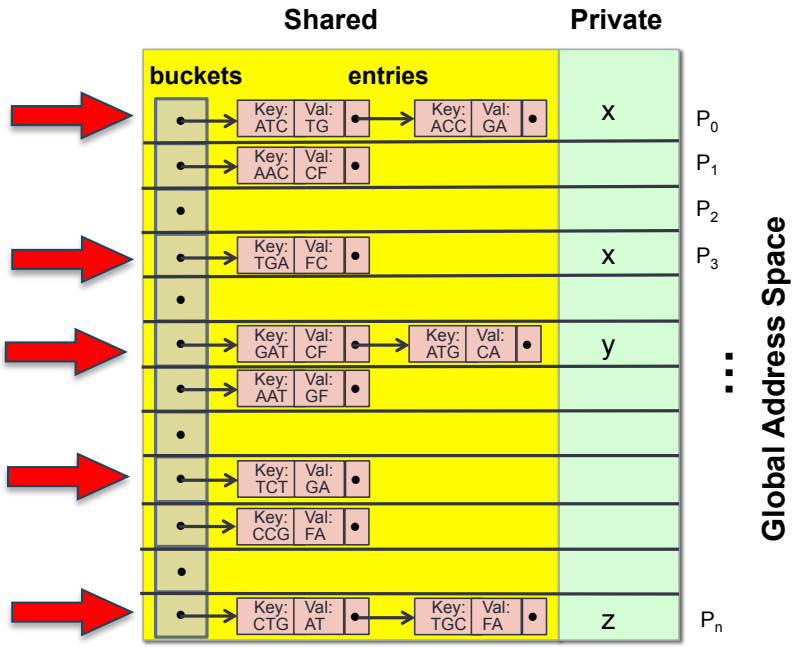
Aggregating stores optimization



Parallel De Bruijn Graph Traversal

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





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:

P_i knows that forward extension is A

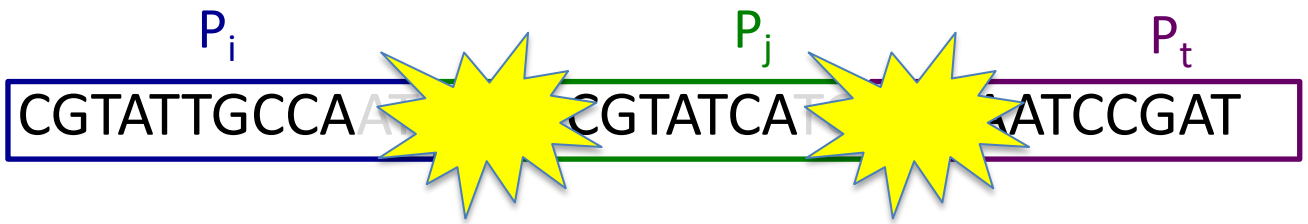
P_i uses the last k-1 bases and the forward extension and forms: CAACGTATCA

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

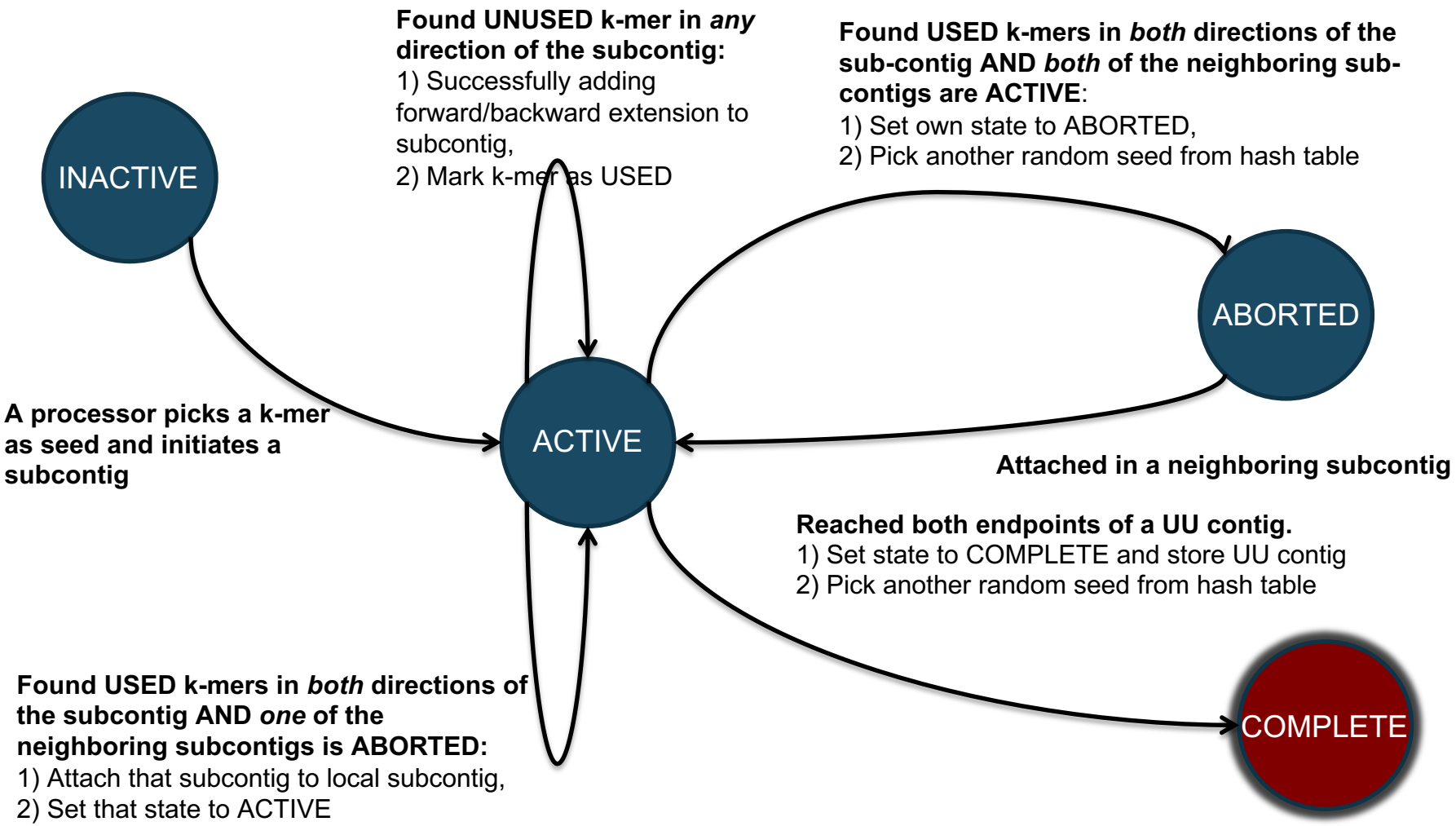
Multiple processors on the same UU contig



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

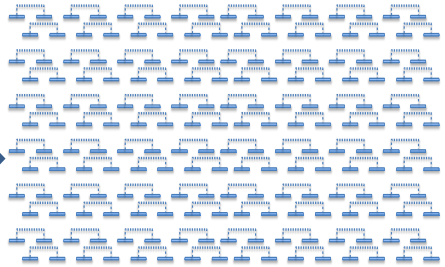
Lightweight Synchronization Protocol



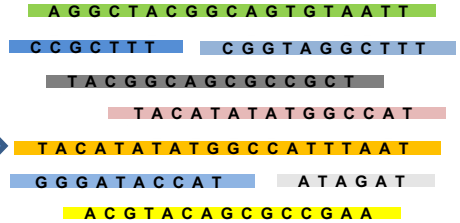
The metagenome assembly problem

Environmental sample
(e.g. bacterial communities)

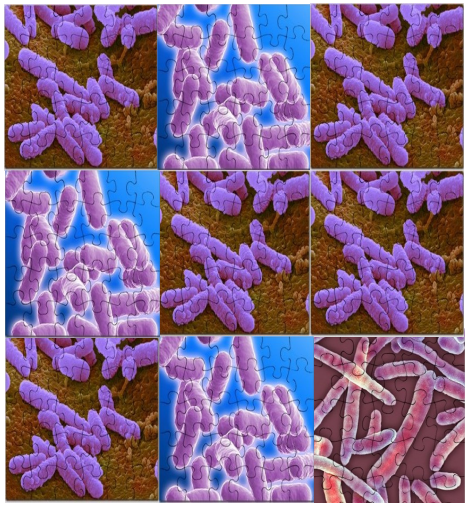
Sequencer

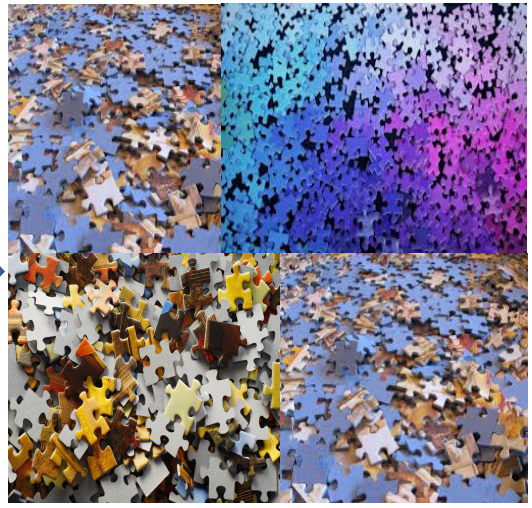


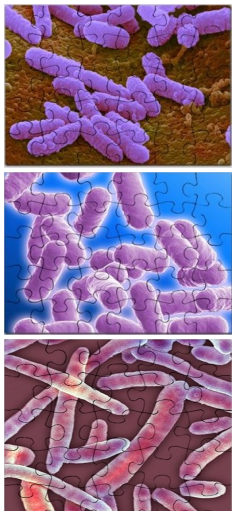
Short “reads” with errors
(100-300 bases long)



Underlying genomes
(up to 10^7 bases long each)



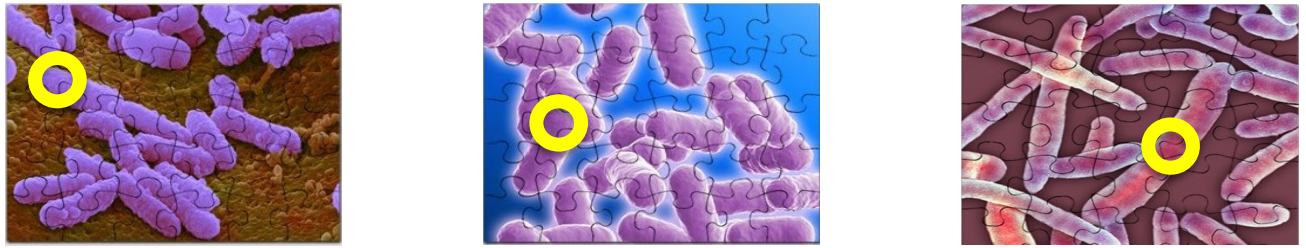




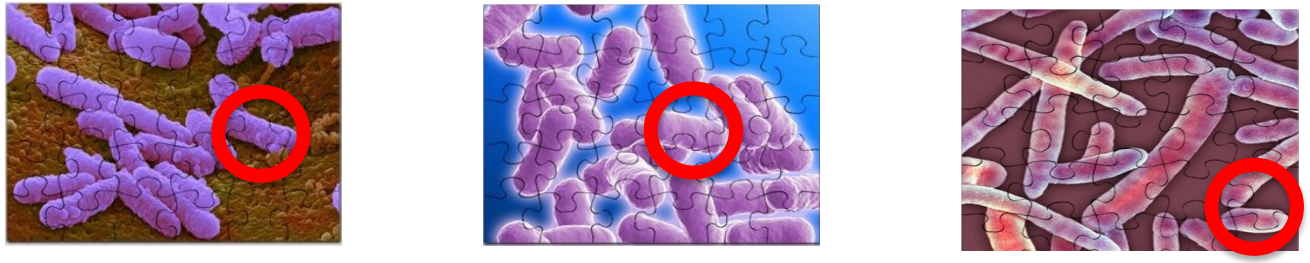
De novo metagenome assembly is harder !!!

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

  - Repeated sequences across genomes



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

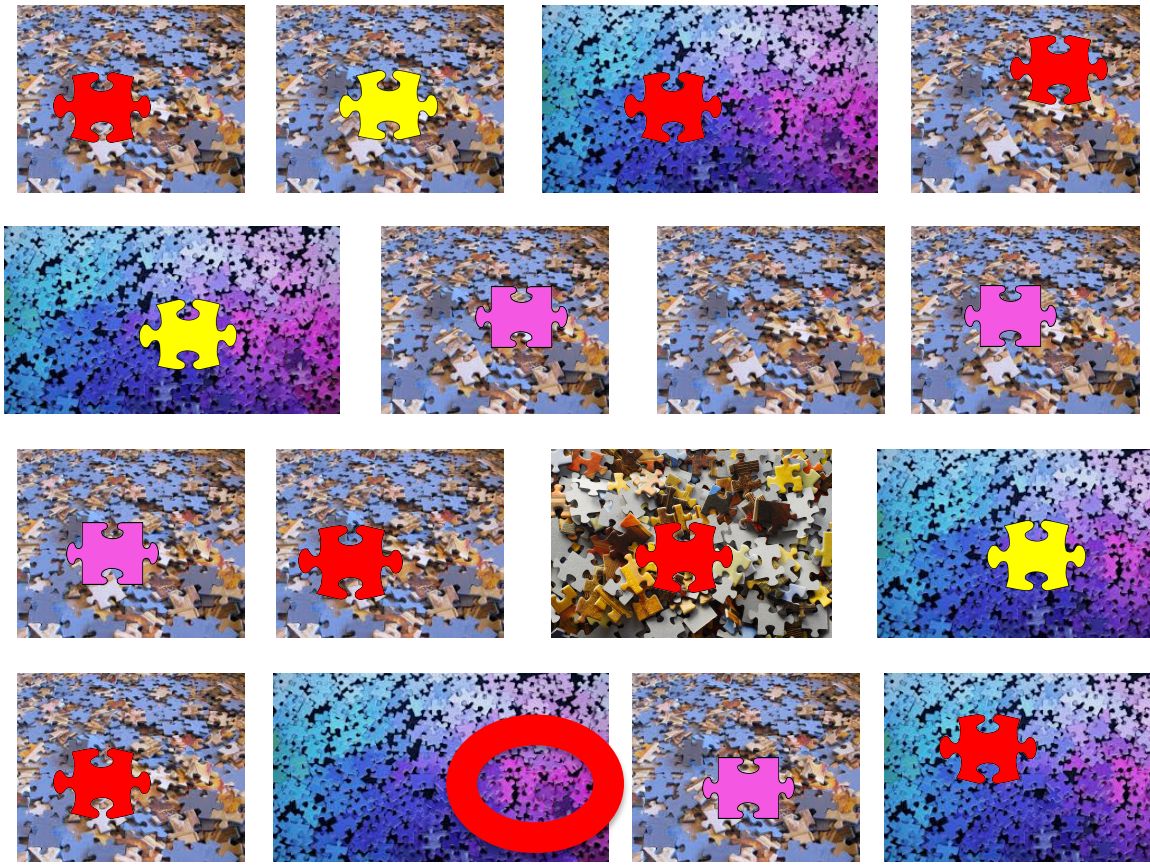


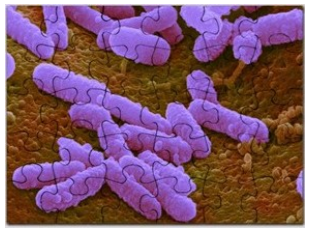
Jigsaw puzzle analogy

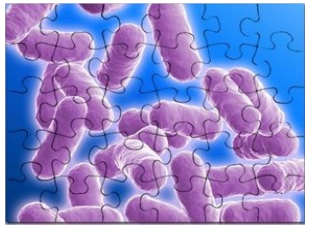
X 10

X 5

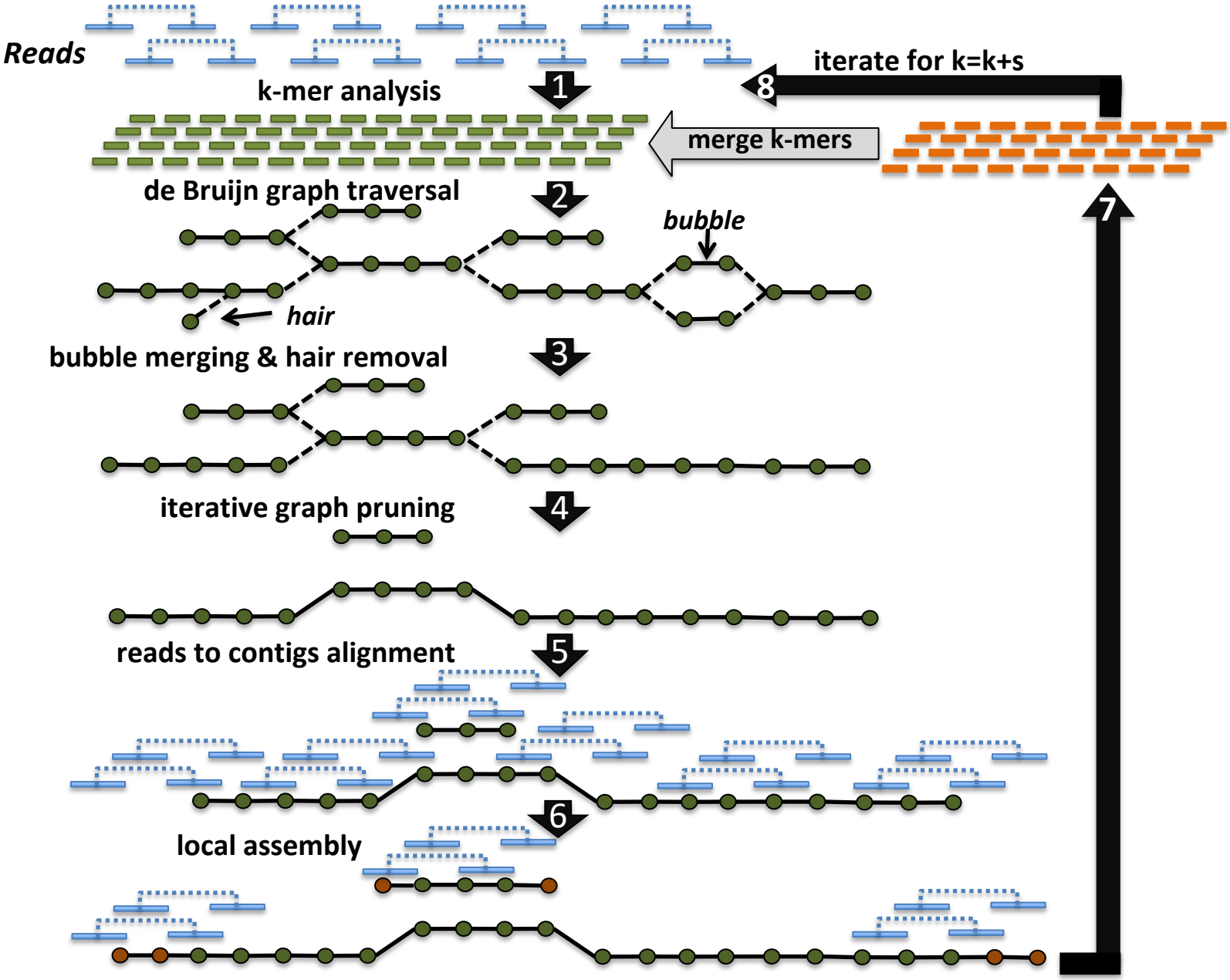
X 1



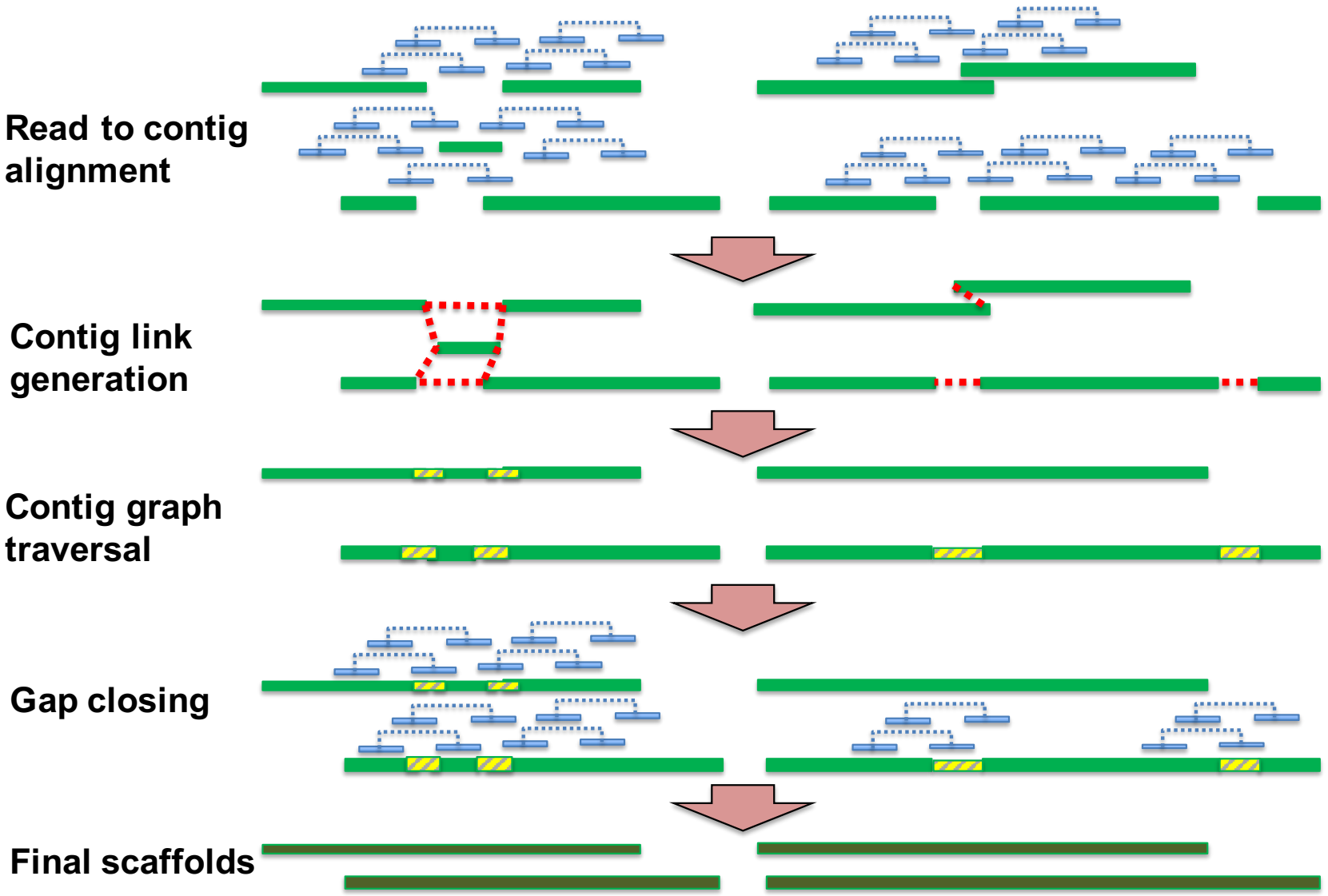




Iterative contig generation algorithm



The MetaHipMer scaffolding algorithm



Parallelization strategy

- The involved algorithms are inherently irregular

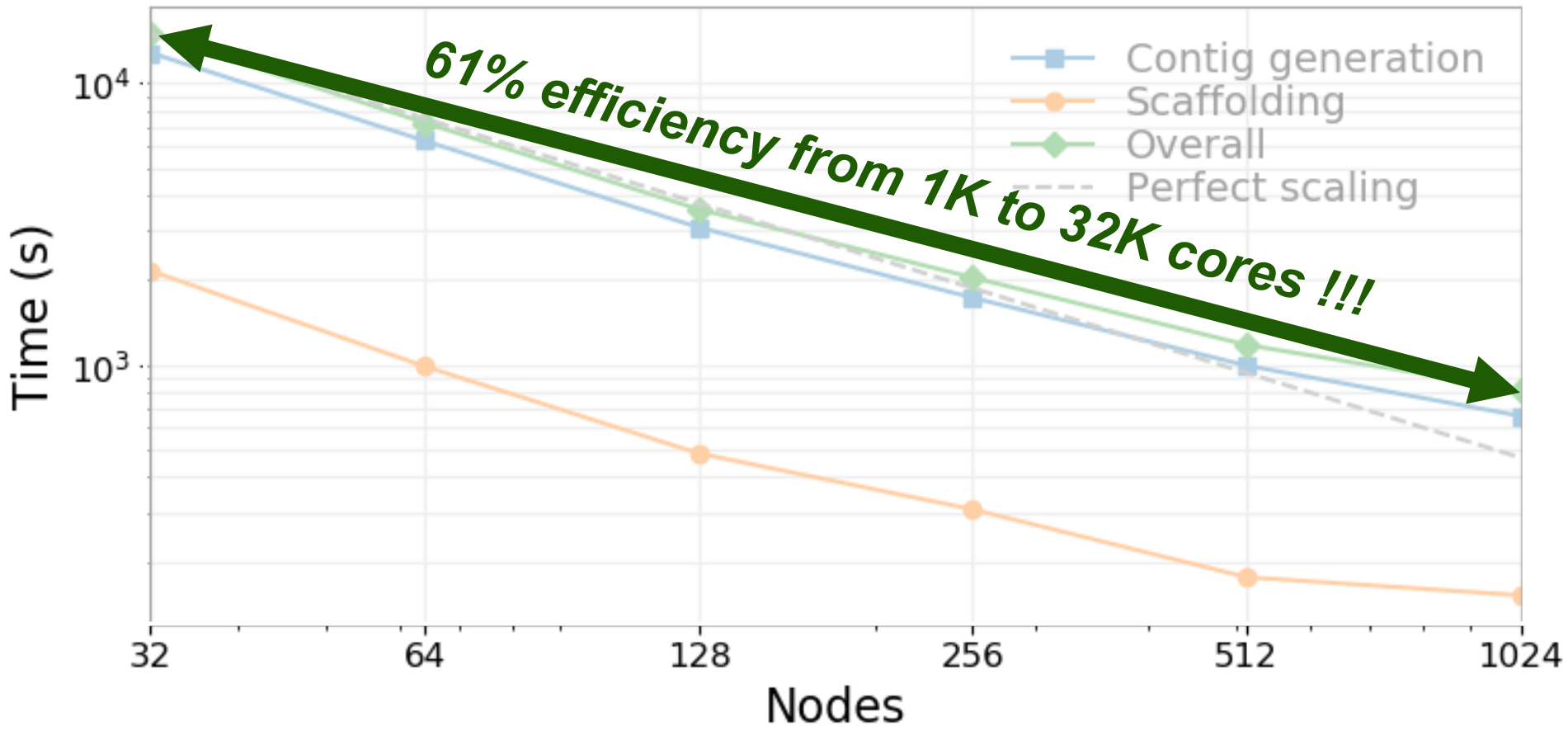
  → irregular & all-to-all communication patterns

- **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 platfrom: 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**

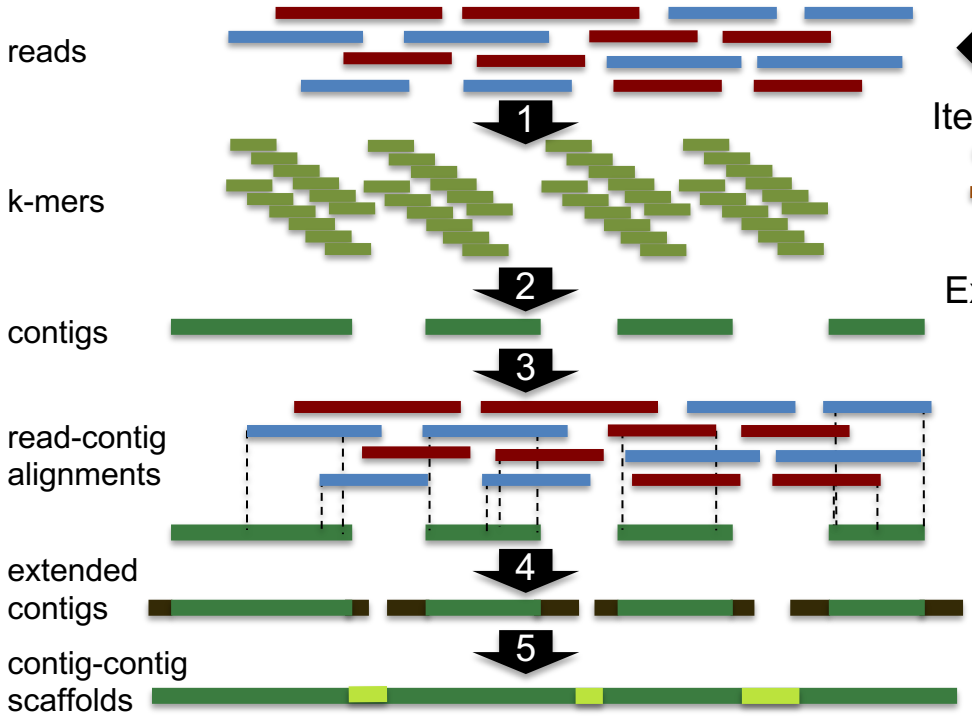
- **MetaHipMer on 512 nodes assembled FULL dataset in 3h25mins**

- This was at the time 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.

Recap

MetaHipmer assembly pipeline for short reads (using UPC++)



1) K-mer Analysis
K-mer histogram

2) Contig Generation
Walk k-mer graph

3) Alignment
Align reads to contigs

4) Local Assembly
Extend ends of contigs

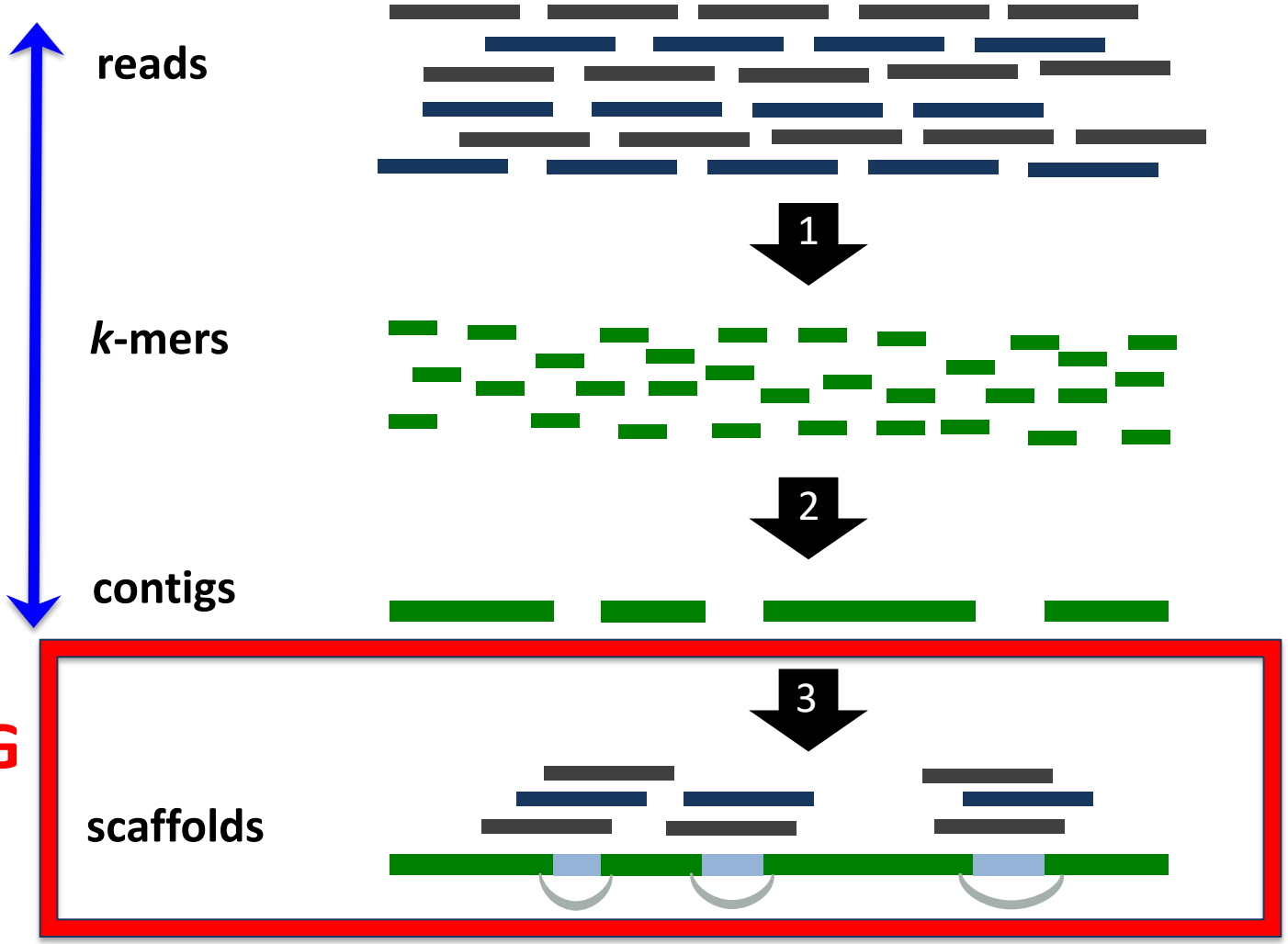
5) Scaffolding
Walk contig graph (iterate)

Outline

- Genome (and metagenome) assembly problems
- **Indexing with hash tables**
- The sparse matrix approach
- Protein Family Identification
- Pairwise sequence alignment
- Indexing with suffix arrays

Alignment for De novo Genome Assembly

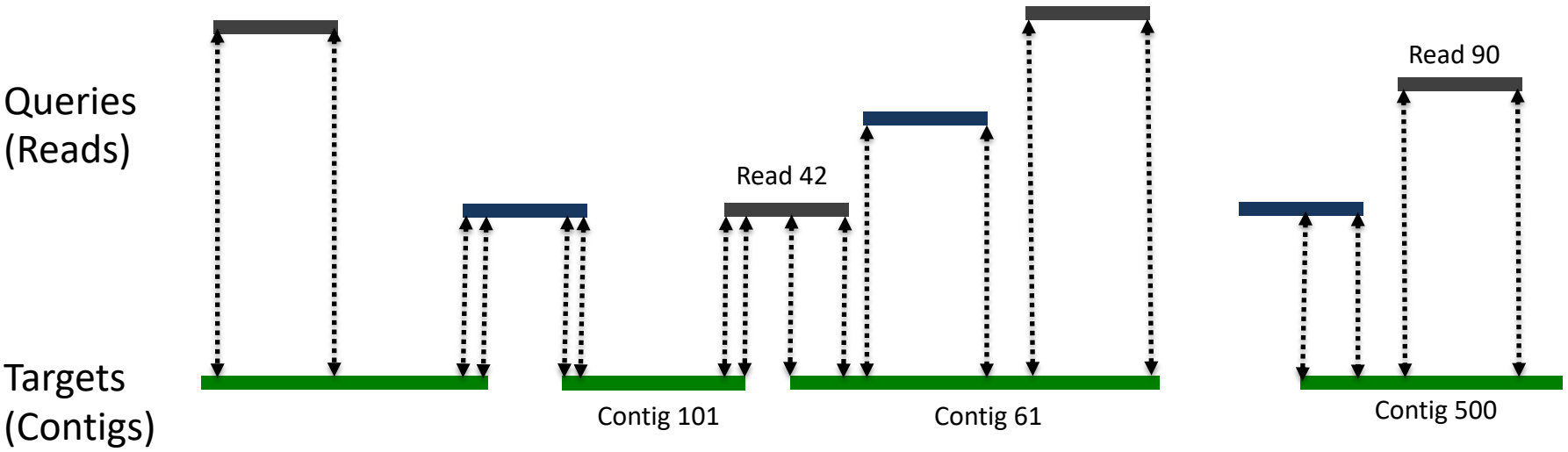
Covered so far



PARALLELIZING SCAFFOLDING

Aligning queries to target sequences

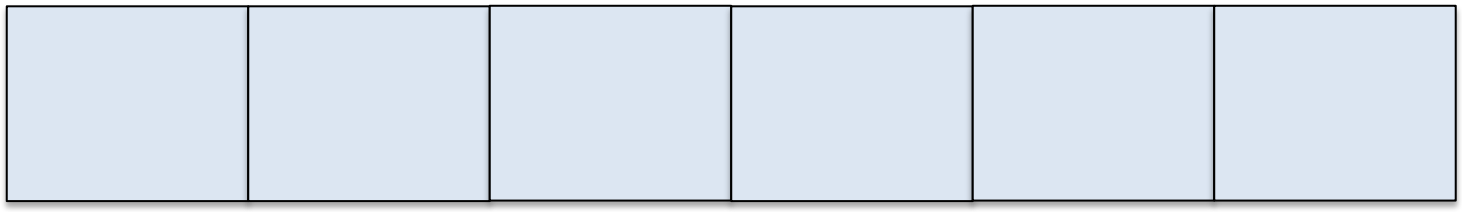
- A query and a target should match in at least **k** bases in order to be aligned
- We call **seed** a substring of a sequence (query or target) with length equal to **k**



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

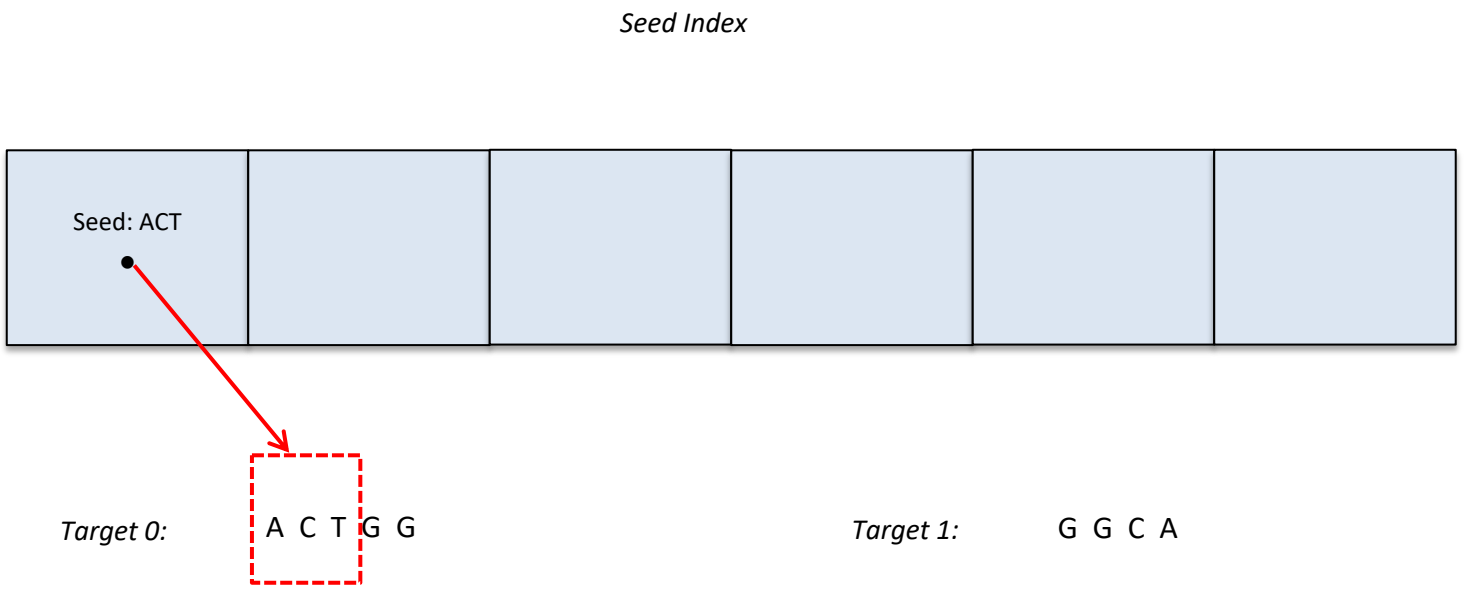
Seed Index



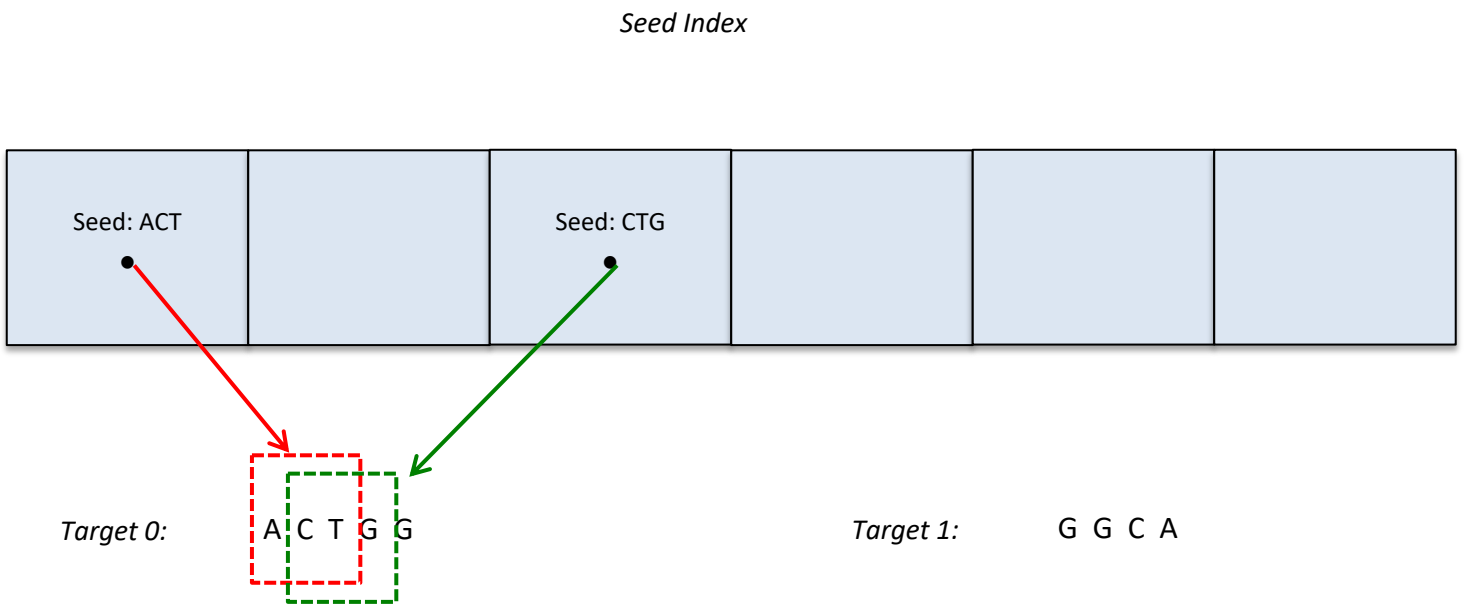
Target 0: A C T G G

Target 1: G G C A

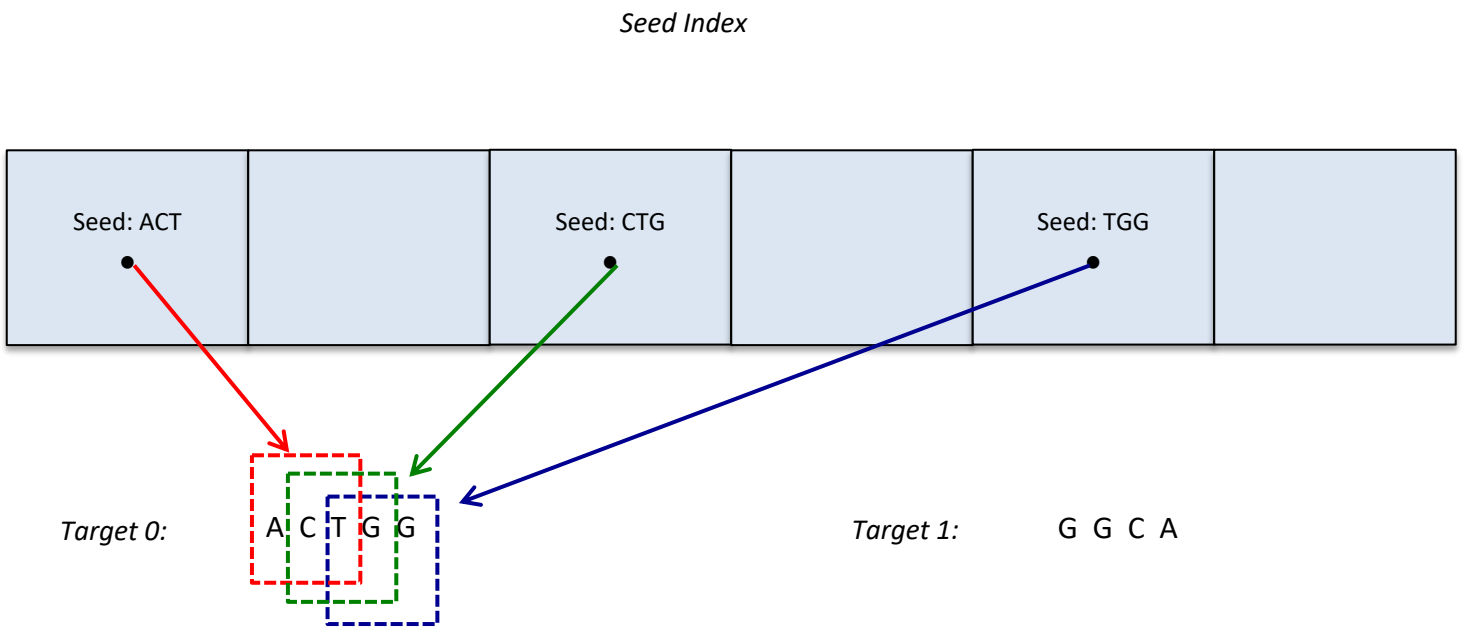
Building seed index



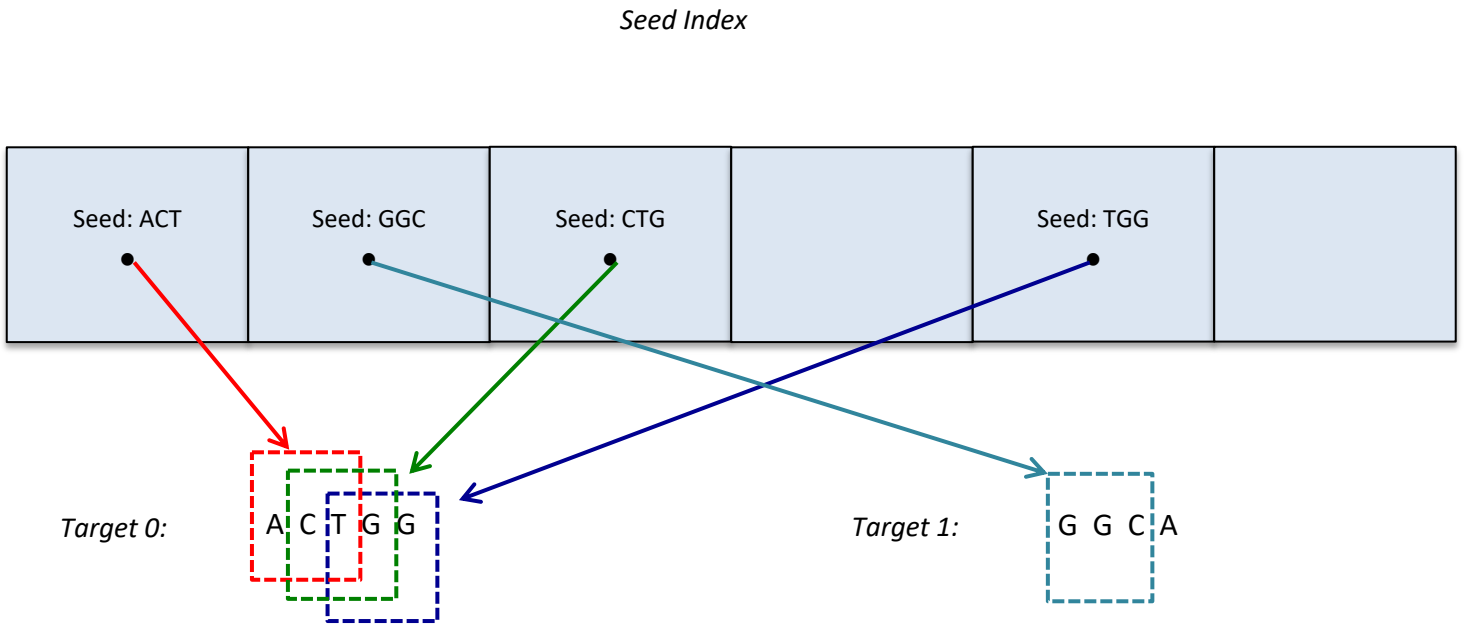
Building seed index



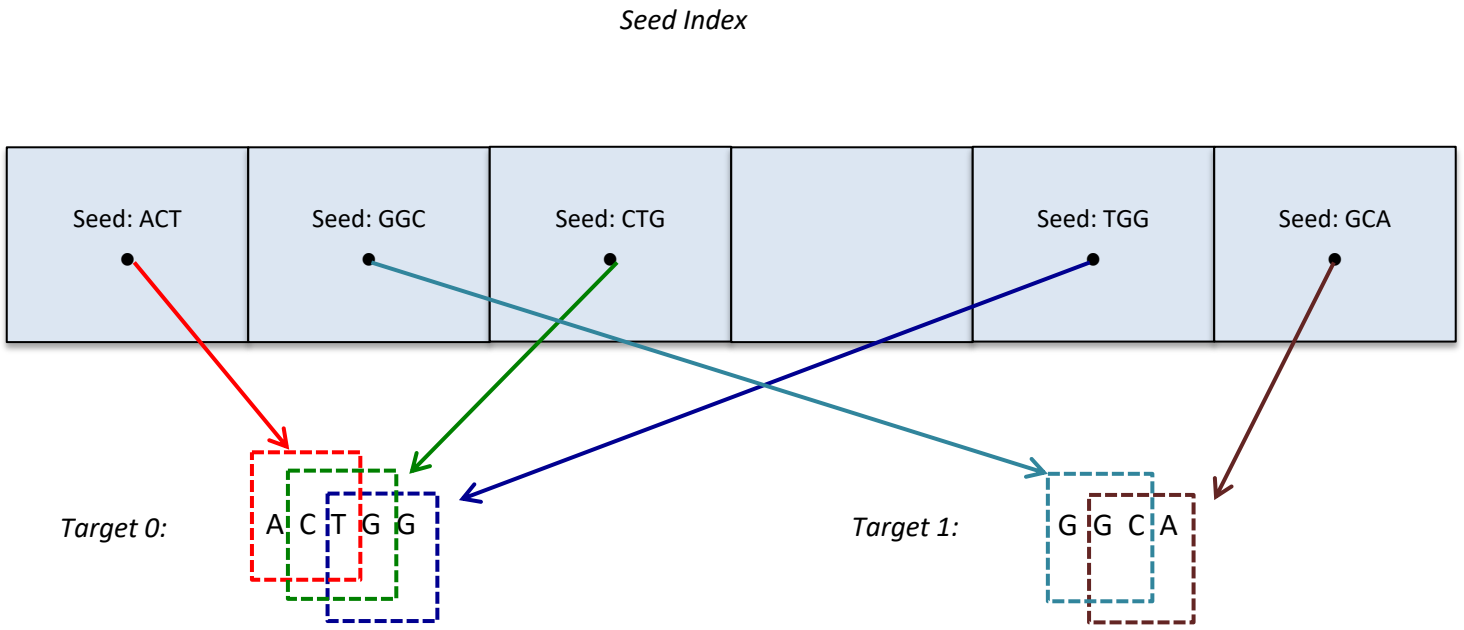
Building seed index



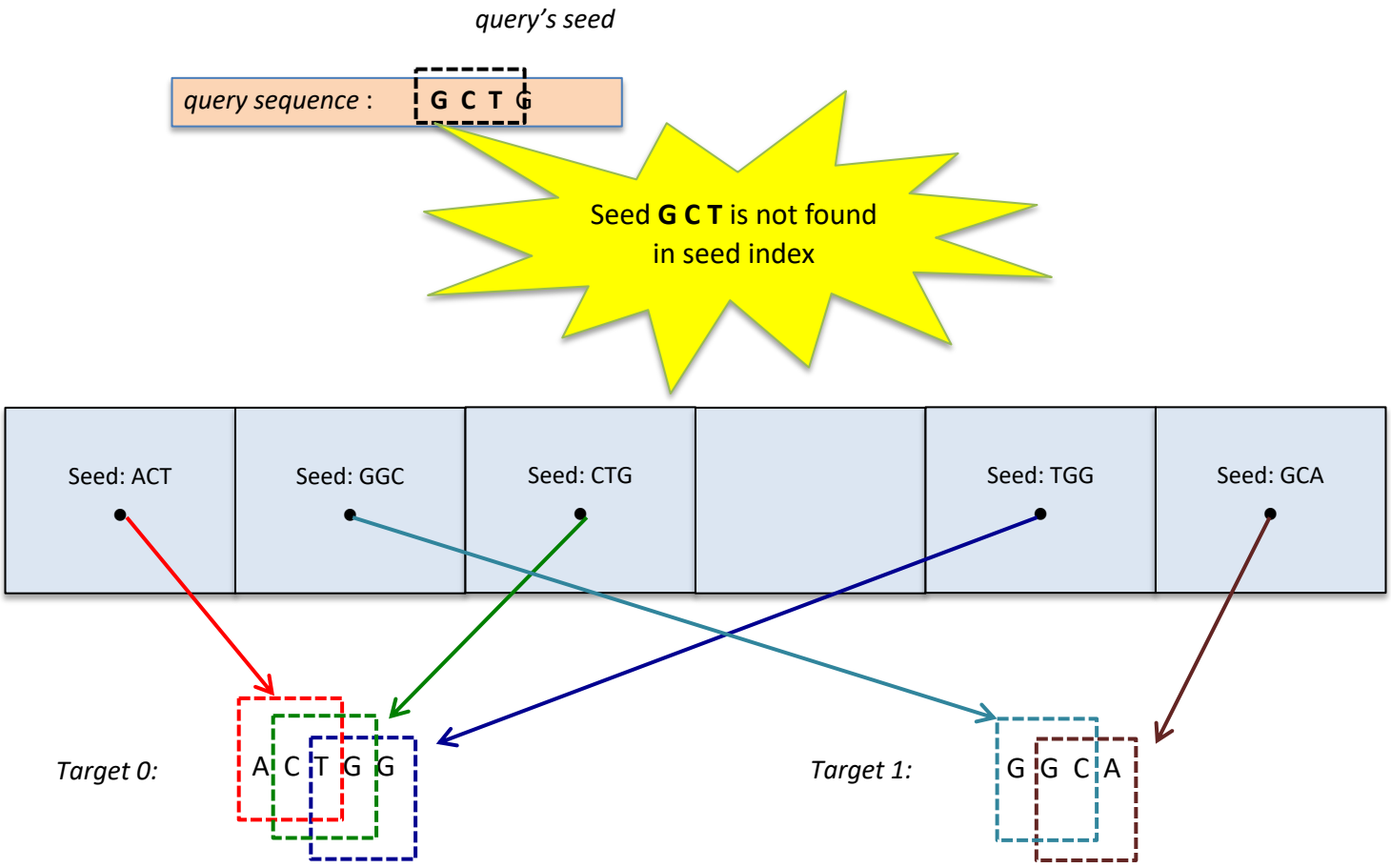
Building seed index



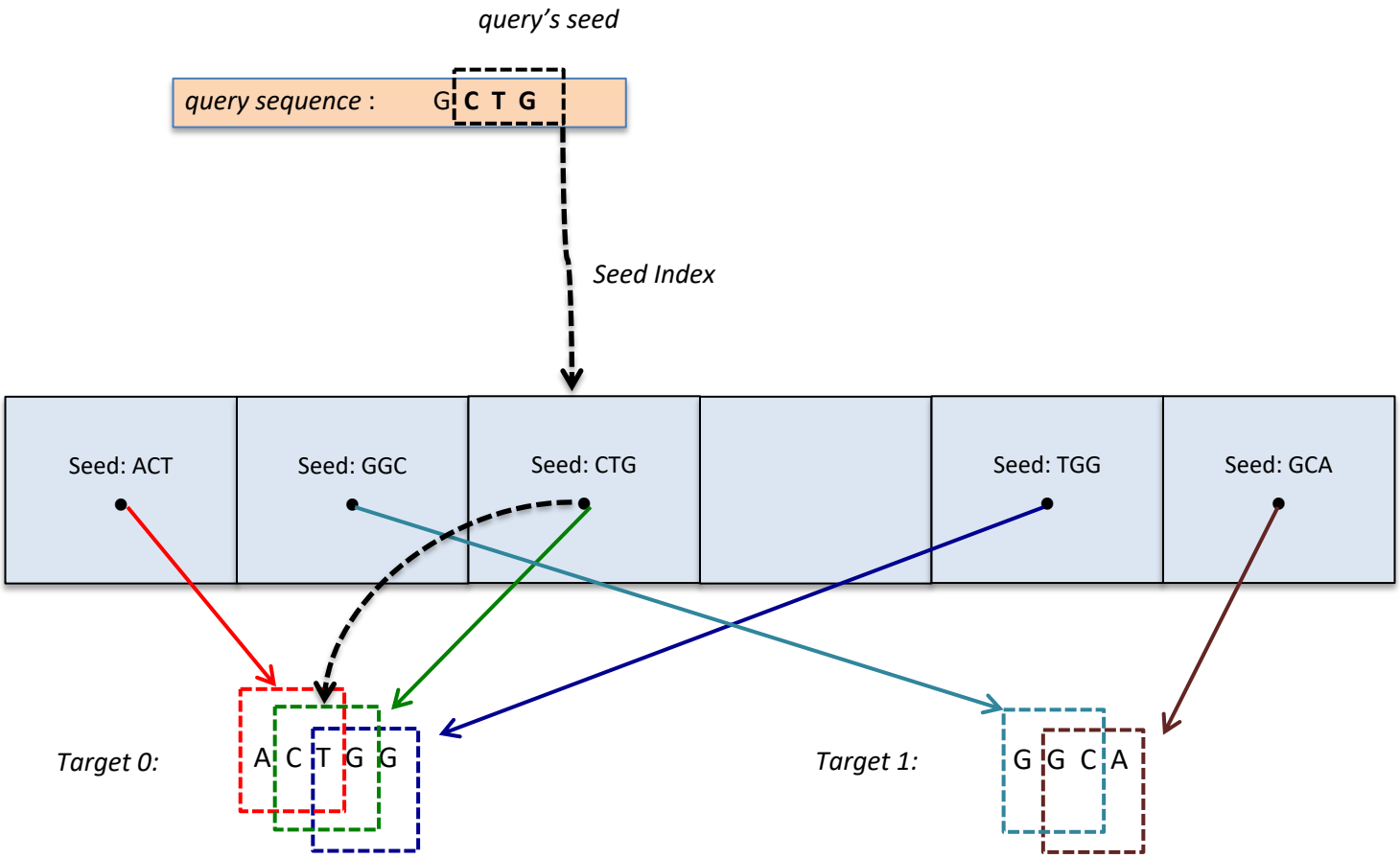
Building seed index



Lookup seed index



Lookup seed index



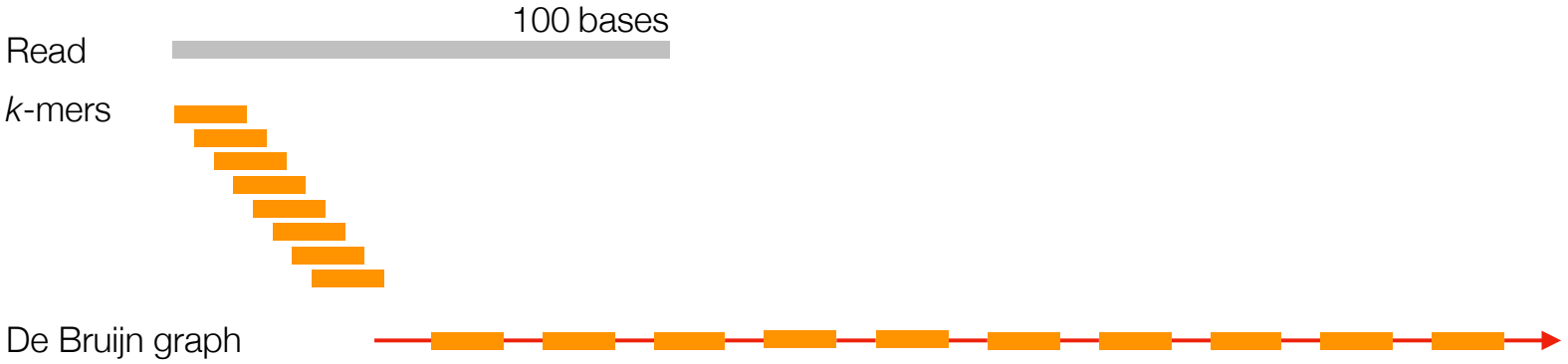
Georganas, E., Buluç, A., Chapman, J., Oliker, L., Rokhsar, D. and Yelick, K. meraligner: A fully parallel sequence aligner. In *IPDPS'15*

Outline

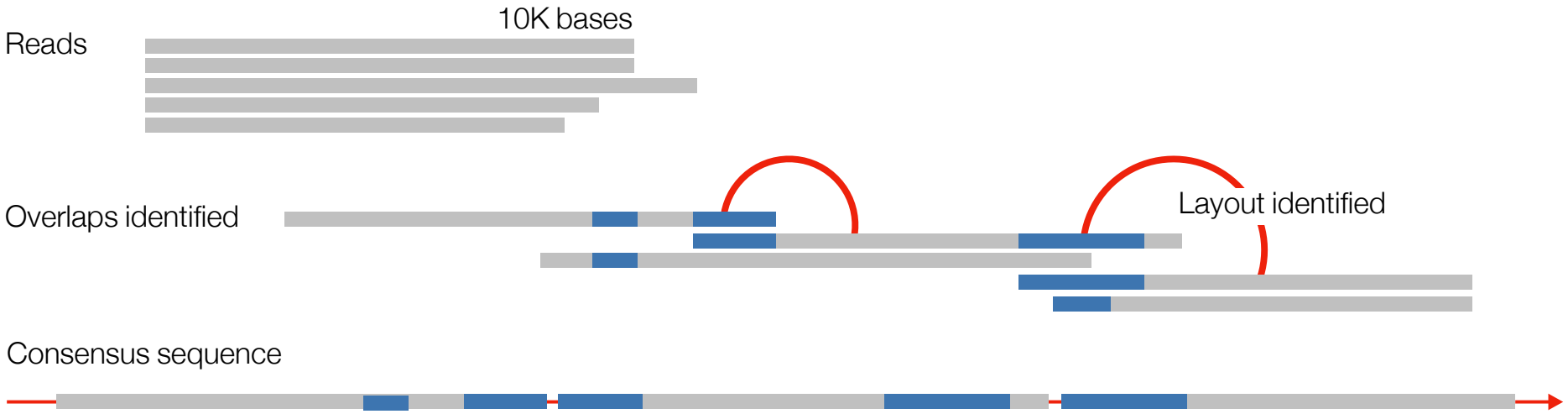
- Genome (and metagenome) assembly problems
- Indexing with hash tables
- **The sparse matrix approach**
- Protein family identification
- Pairwise sequence alignment
- Indexing with suffix arrays

Alternative assembly paradigms

De Bruijn Graph



Overlap-Layout-Consensus

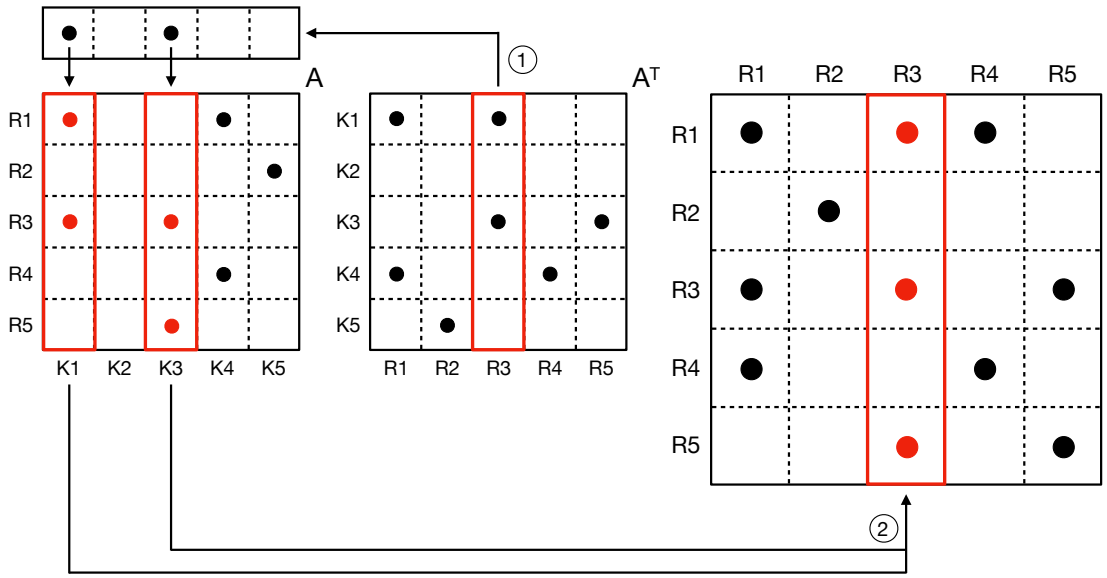


Assembly with Long Noisy Reads

- **Long reads** from PacBio and Oxford Nanopore have been revolutionizing de-novo assembly
- Error rates are higher at 1-15%
- 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: majority of k-mers are wrong

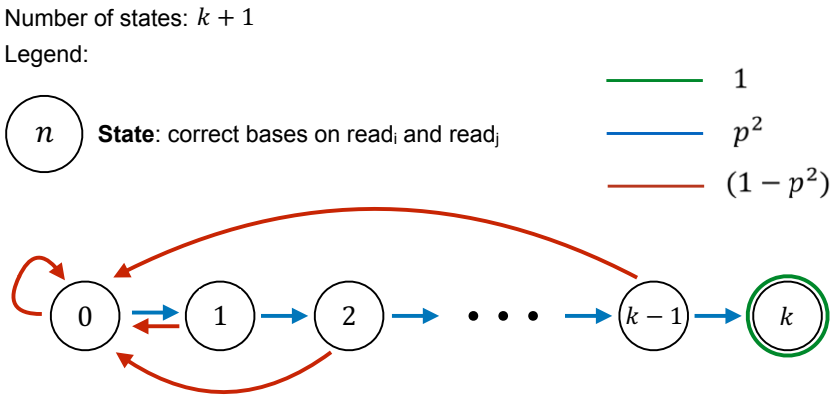
Candidate alignments among reads

- **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

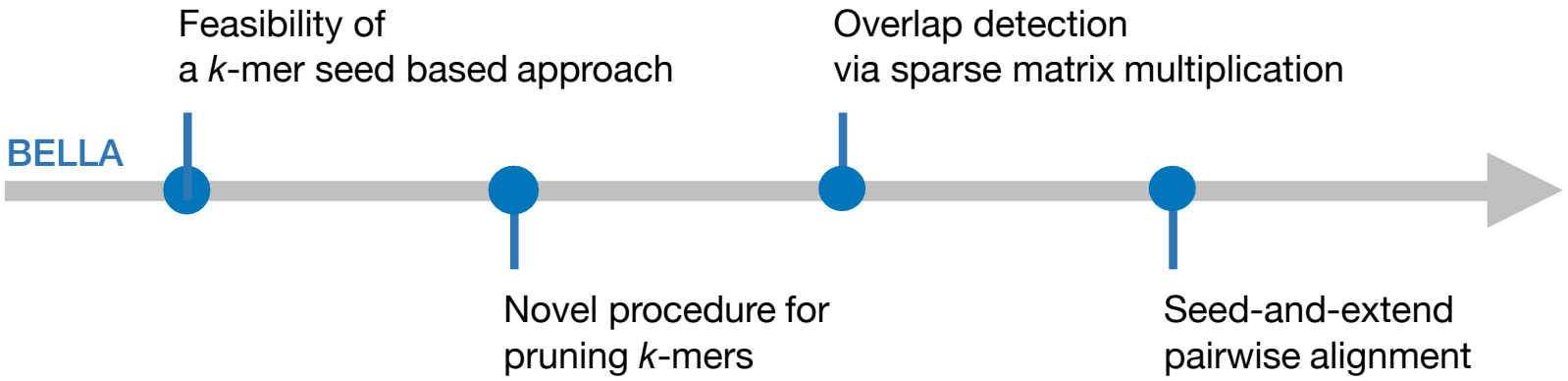


Read overlapping using shared k-mers is computing a **sparse matrix product**, for which we know good algorithms and implementations

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



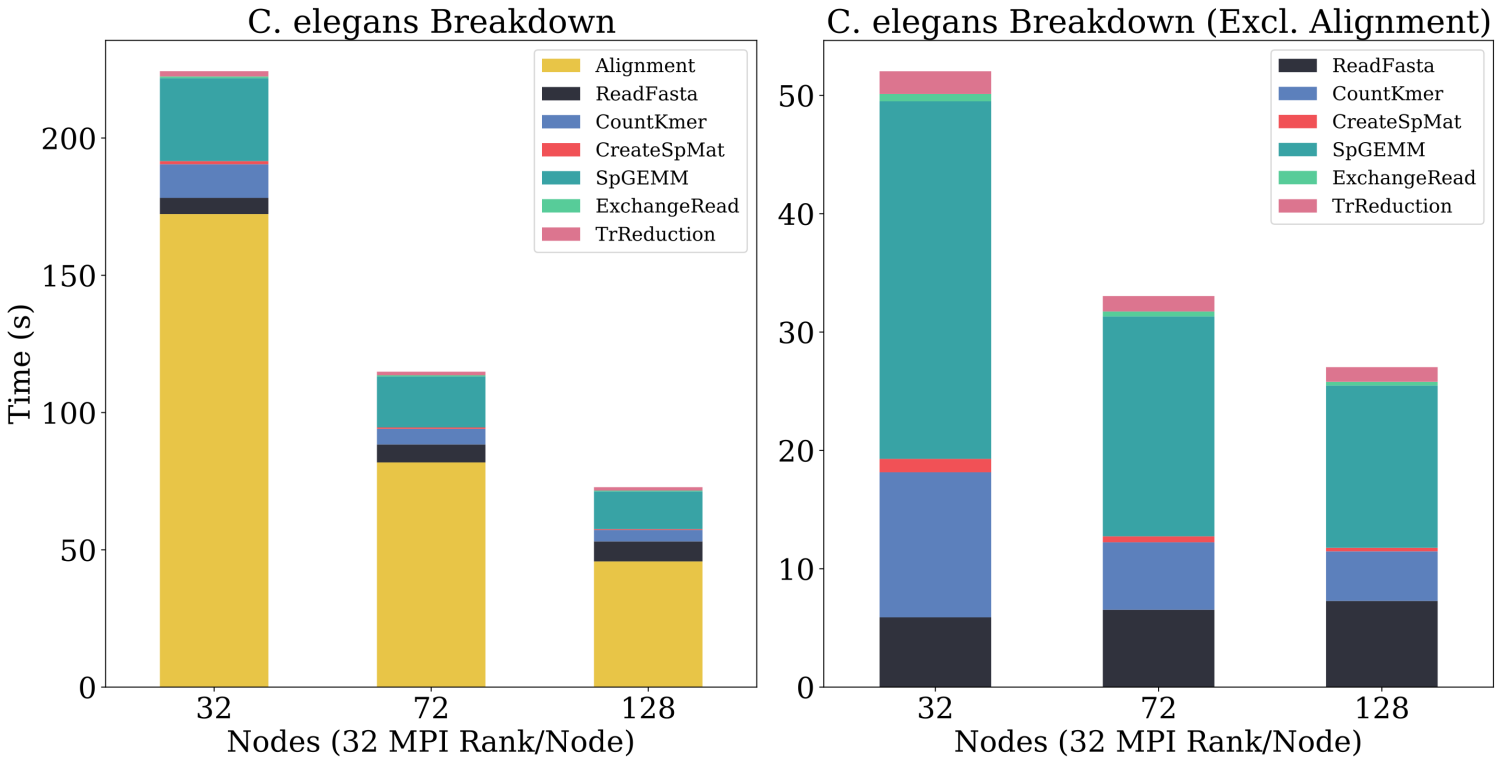
- How to choose the right set of k-mers, otherwise there are too many of them?
- How to use alignment score to tell true alignments from false positives?



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

diBELLA.2D performance results

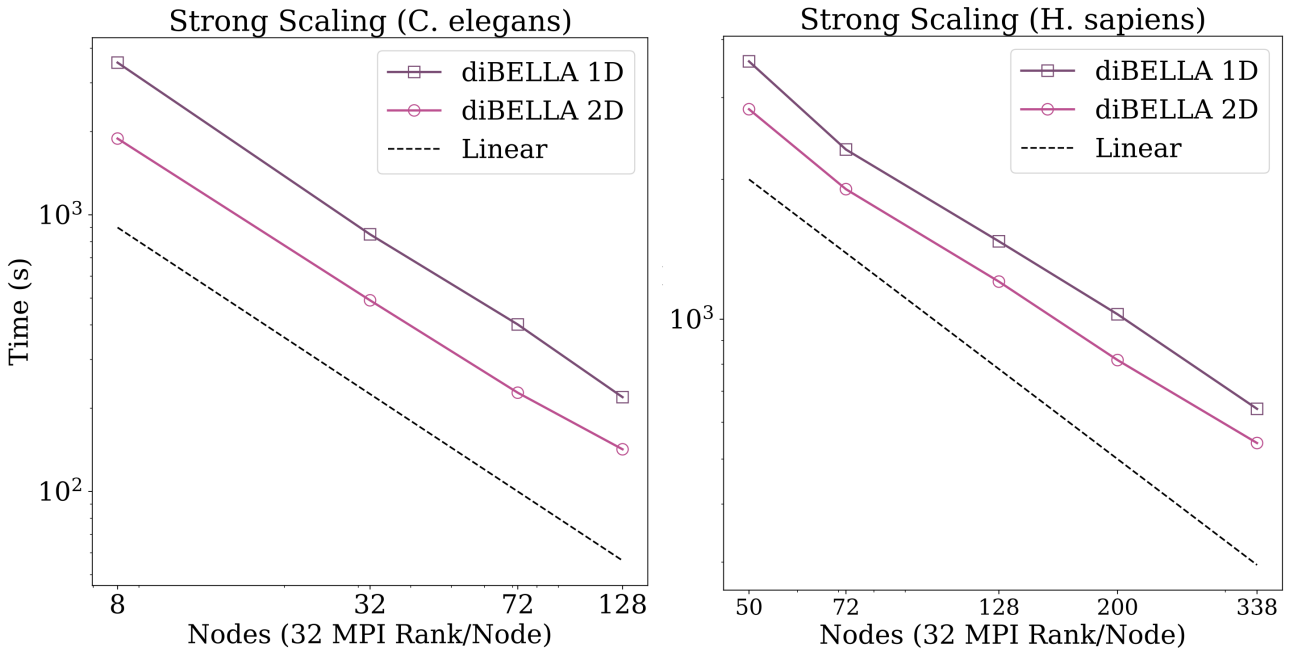
diBELLA.2D: distributed-memory version of BELLA on 2D process grid
Performs *overlap detection* plus *transitive reduction (overlap to string graph)*
https://github.com/PASSIONLab/diBELLA.2D



Giulia Guidi, Oguz Selvitopi, Marquita Ellis, Leonid Oliker, Katherine Yelick, Aydin Buluç. Parallel String Graph Construction and Transitive Reduction for De Novo Genome Assembly. *IPDPS* 2021

Is the sparse matrix approach better?

- Comparing the sparse matrix abstraction (diBELLA 2D [2], **weeks** of effort) with a direct implementation (diBELLA 1D [1], **years** of effort). Both use MPI
- Sparse matrices reduce communication via 2D sparse SpGEMM

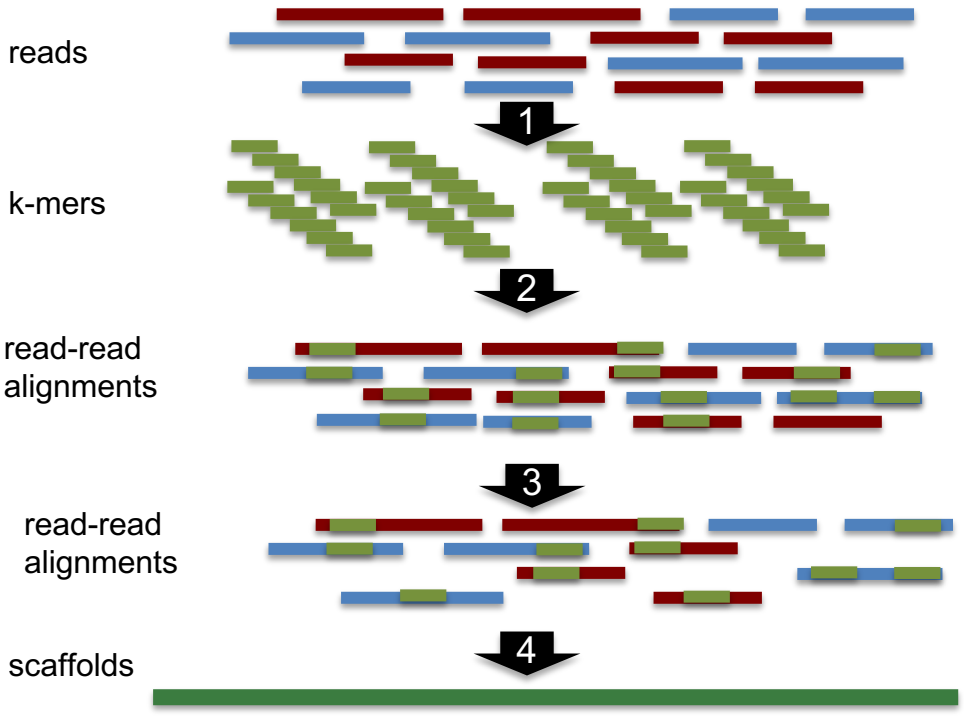


[1] Marquita Ellis, Giulia Guidi, Aydin Buluç, Leonid Oliker, and Katherine Yelick. "diBELLA: Distributed long read to long read alignment." ICPP 2019

[2] Giulia Guidi, Oguz Selvitopi, Marquita Ellis, Leonid Oliker, Katherine Yelick, Aydin Buluç. Parallel String Graph Construction and Transitive Reduction for De Novo Genome Assembly. *IPDPS* 2021

Recap

Sparse matrix (diBELLA.2D) approach for assembly with long reads



1) K-mer Analysis
K-mer histogram

2) Sparse matrix building
A: reads-by-kmers

3) Overlapping via SpGEMM
$C = AA^T$: reads-by-reads

4) X-drop alignments
M = filter(C, alignment_score)

5) Transitive Reduction
$M^{i+1} = \text{Prune}(M^i M^i \odot M^i)$

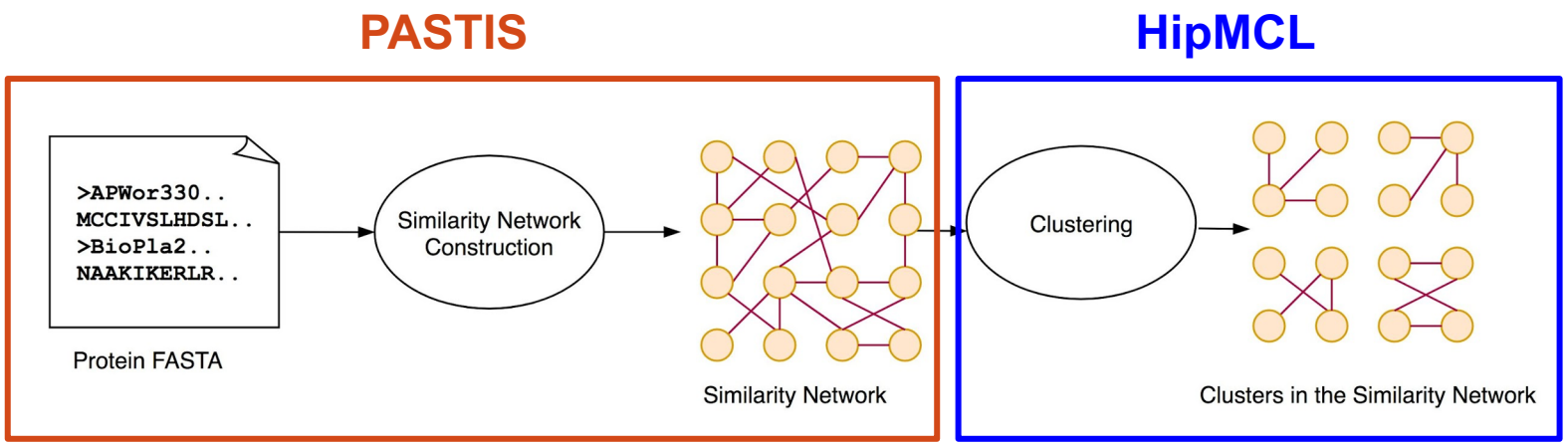
diBELLA is not yet a full assembler and does not support metagenomes

Outline

- Genome (and metagenome) assembly problems
- Indexing with hash tables
- The sparse matrix approach
- **Protein family identification**
- Pairwise sequence alignment
- Indexing with suffix arrays

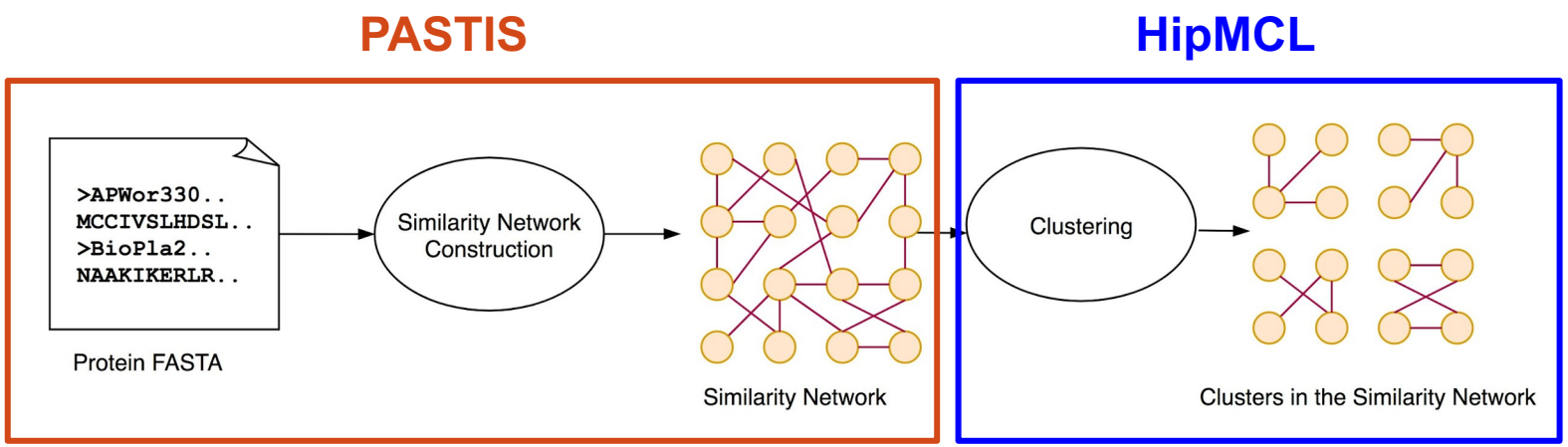
Protein Family Identification

- **Problem:** Given a large collection of proteins, identify groups of proteins that are homologous (i.e. descended from a common ancestor).
- Homologous proteins often have the same function (think of different variants of hemoglobin in many species)
- Often, only sequences (and not structure) of the proteins are available, so we infer homology via sequence similarity



Protein Family Identification

- Many one-step approaches are possible that trade accuracy for lower memory consumption and faster execution (e.g., CD-HIT).
- The approach that seems to lead to highest accuracy:
  - Construct a **similarity network** over protein sequences using many-to-many sequence search (PASTIS)
  - Cluster this network to discover possible protein families (HipMCL)



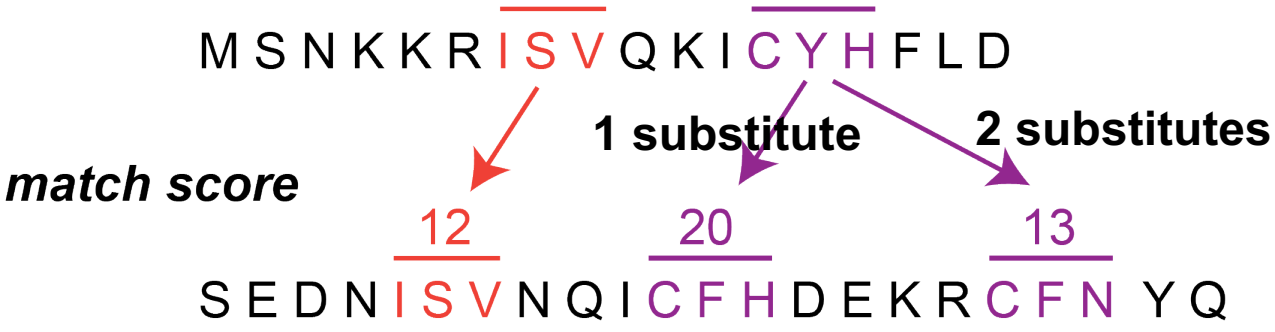
SpGEMM for many-to-many protein alignment

- Idea similar to BELLA, but removing the exact match restriction
- For homology detection, need to catch weaker signal (~30% ANI)
- K-mers with substitutes may be more valuable than exact matches!

BLOSUM 62 scoring matrix

(positive values are shaded)

	A	R	N	D	C	Q	E	G	H	I	L	K	M	F	P	S	T	W	Y	V
A	4																			
R	-1	5																		
N	-2	0	6																	
D	-2	-2	1	6																
C	0	-3	-3	-3	9															
Q	-1	1	0	0	-3	5														
E	-1	0	0	2	-4	2	5													
G	0	-2	0	-1	-3	-2	-2	6												
H	-2	0	1	-1	-3	0	0	-2	8											
I	-1	-3	-3	-3	-1	-3	-3	-4	-3	4										
L	-1	-2	-3	-4	-1	-2	-3	-4	-3	2	4									
K	-1	2	0	-1	-3	1	1	-2	-1	-3	-2	5								
M	-1	-1	-2	-3	-1	0	-2	-3	-2	1	2	-1	5							
F	-2	-3	-3	-3	-2	-3	-3	-3	-1	0	0	-3	0	6						
P	-1	-2	-2	-1	-3	-1	-1	-2	-2	-3	-3	-1	-2	-4	7					
S	1	-1	1	0	-1	0	0	0	-1	-2	-2	0	-1	-2	-1	4				
T	0	-1	0	-1	-1	-1	-1	-2	-2	-1	-1	-1	-1	-2	-1	1	5			
W	-3	-3	-4	-4	-2	-2	-3	-2	-2	-3	-2	-3	-1	1	-4	-3	-2	11		
Y	-2	-2	-2	-3	-2	-1	-2	-3	2	-1	-1	-2	-1	3	-3	-2	-2	2	7	
V	0	-3	-3	-3	-1	-2	-2	-3	-3	3	1	-2	1	-1	-2	-2	0	-3	-1	4



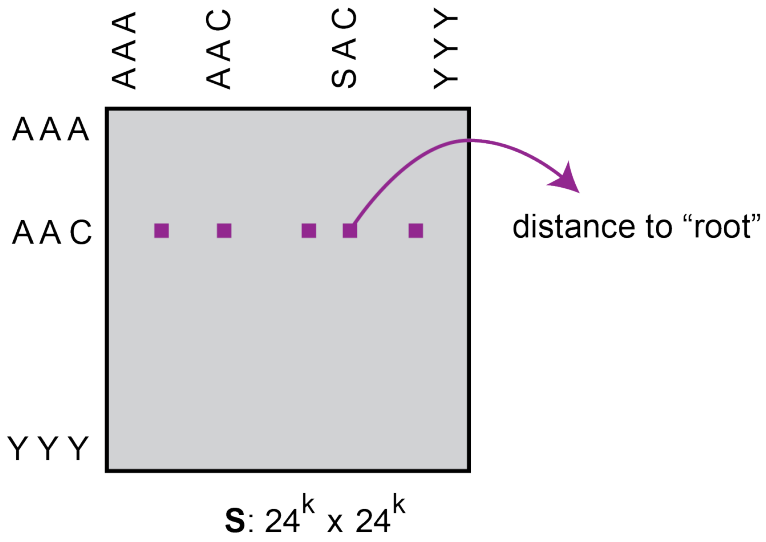
SpGEMM for many-to-many protein alignment

PASTIS (https://github.com/PASSIONLab/PASTIS) does distributed many-to-many protein sequence similarity search using sparse matrices

Introduce new sparse matrix **S**

Contains substitution information

Each entry has **substitution cost**



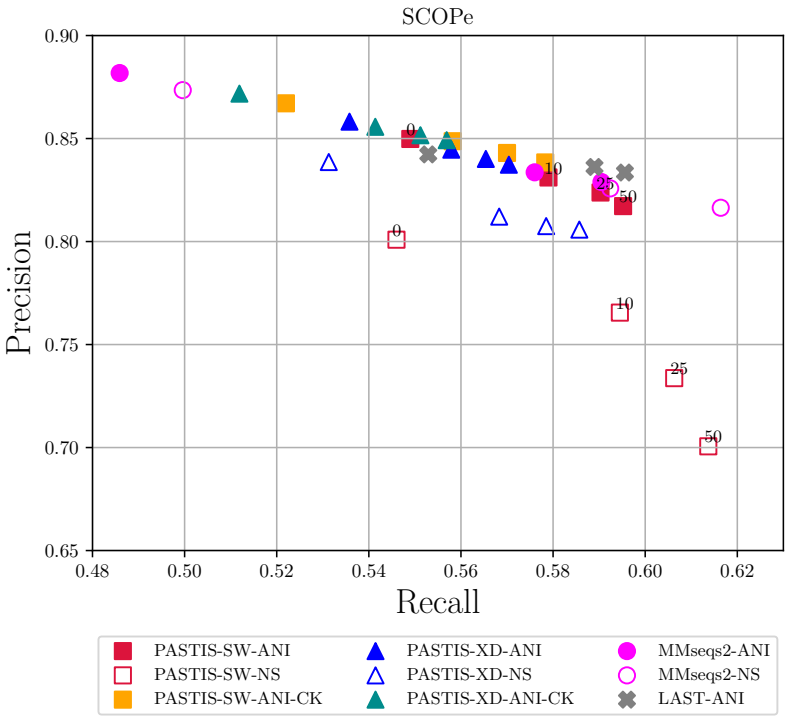
Exact k-mers → $C=AA^T$

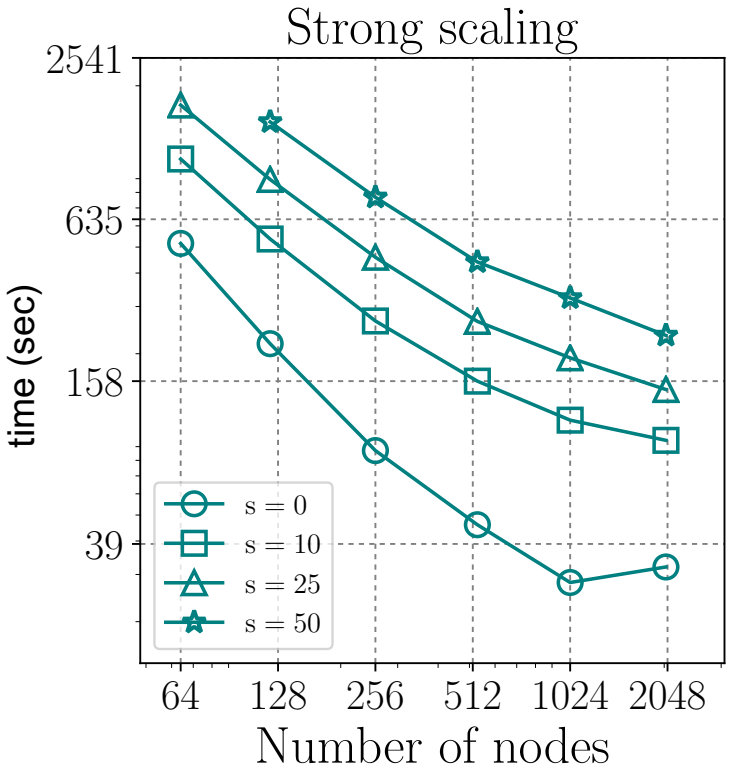
Substitute k-mers → $C=ASA^T$

New semiring

Oguz Selvitopi, Saliya Ekanayake, Giulia Guidi, Georgios Pavlopoulos, Ariful Azad, and Aydın Buluç. Distributed Many-to-Many Protein Sequence Alignment Using Sparse Matrices. SC'20.

PASTIS performance and accuracy

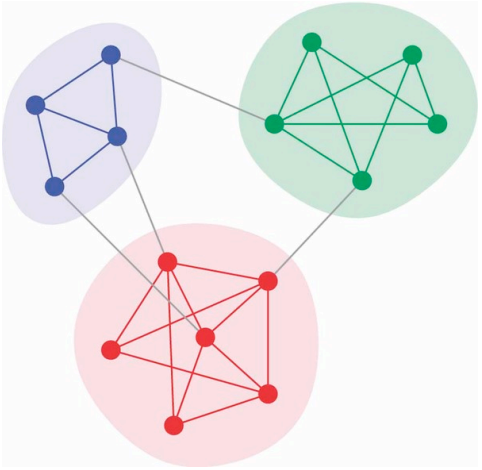




- *Protein similarity search* is the first and most time-consuming step in discovering protein families (proteins evolved from a common ancestor and who likely have the same function)
- *Protein family identification* is a key step in protein function discovery and taxonomic assignment of newly sequenced organisms

The Markov Cluster Algorithm (MCL)

Widely popular and successful algorithm for discovering clusters (e.g. protein families) in protein interaction and protein sequence similarity networks

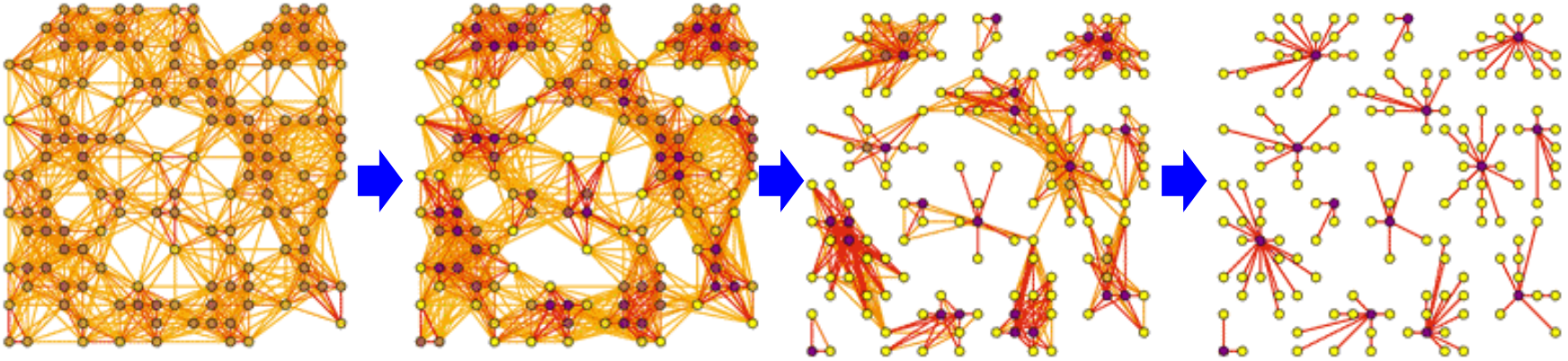


The number of **edges or higher-length paths** between two arbitrary nodes in a cluster is greater than the number of paths between nodes from different clusters

Random walks on the graph will frequently remains within a cluster

The algorithm **computes the probability** of random walks through the graph and **removes lower probability terms** to form clusters

The Markov Cluster Algorithm (MCL)



Initial network | Iteration 1 | Iteration 2 | Iteration 3

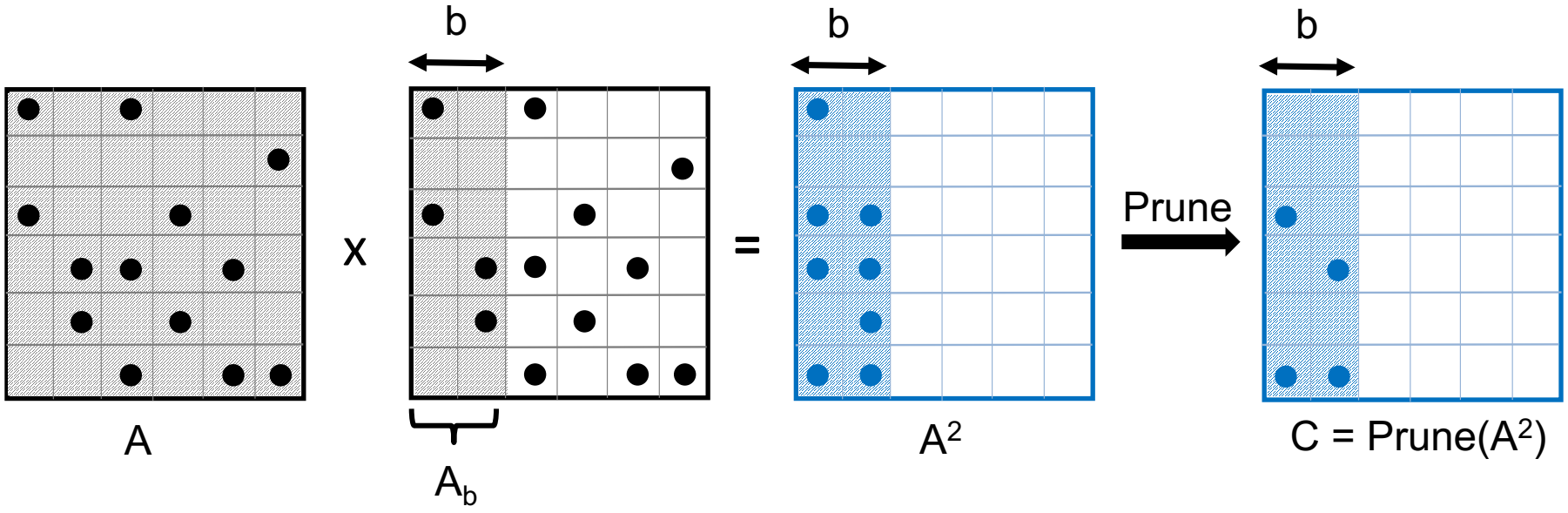
At each iteration:

Step 1 (Expansion): Squaring the matrix while pruning (a) small entries, (b) denser columns

Naïve implementation: sparse matrix-matrix product (SpGEMM), followed by column-wise top-K selection and column-wise pruning

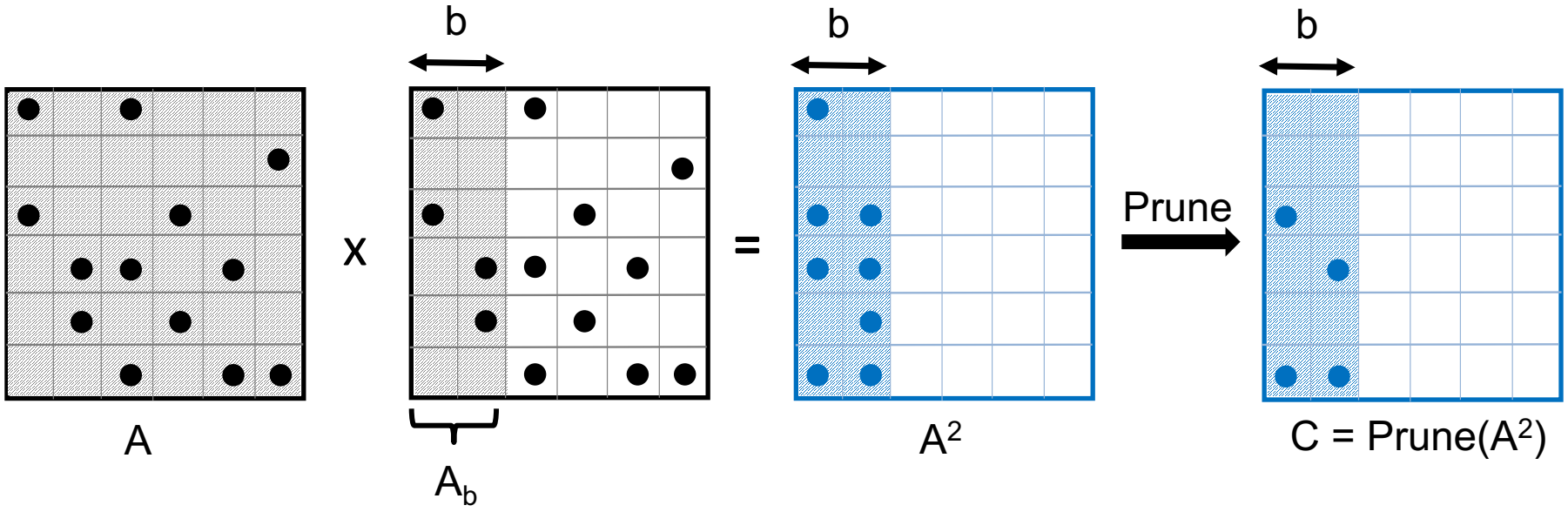
Step 2 (Inflation) : taking powers entry-wise

A combined expansion and pruning step



- b: number of columns in the output constructed at once
  - Smaller b: less parallelism, memory efficient (b=1 is equivalent to sparse matrix-sparse vector multiplication used in MCL)
  - Larger b: more parallelism, memory intensive

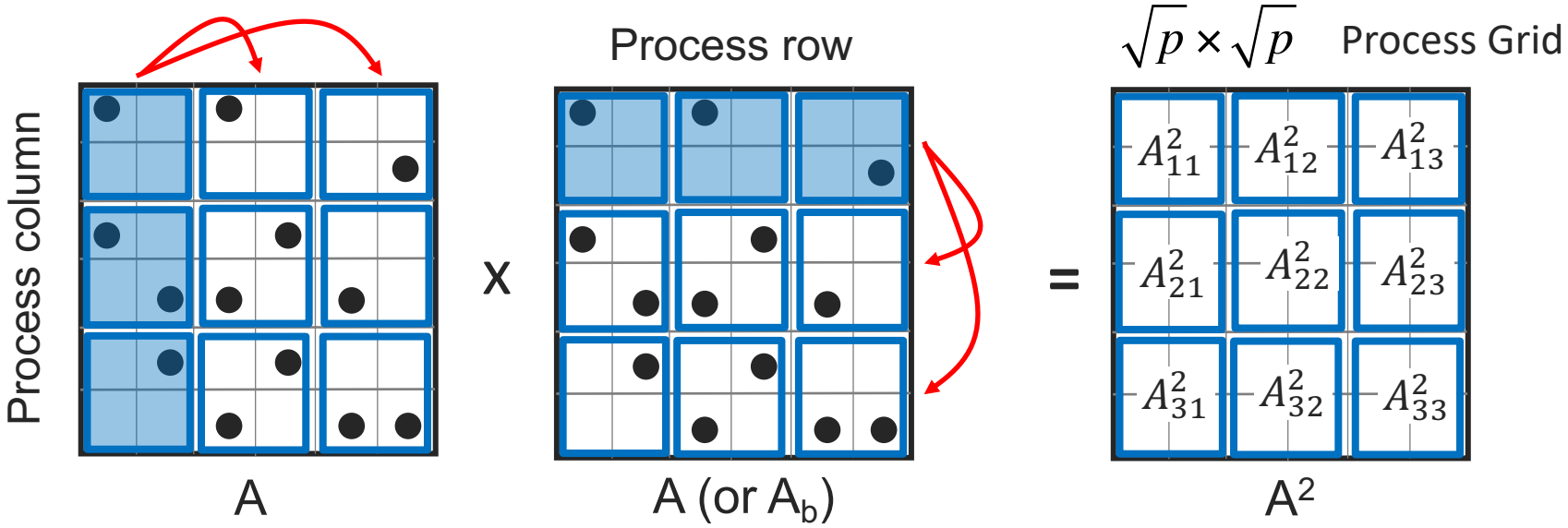
A combined expansion and pruning step



- b: number of columns in the output constructed at once
  - HipMCL selects b dynamically as permitted by the available memory
  - The algorithm works in h=N/b phases where N is the number of columns (vertices in the network) in the matrix

HipMCL: High-performance MCL

- HipMCL uses the most popular variant of Sparse SUMMA
- Both input matrices are broadcasted in stages and owners of output submatrices perform local sparse matrix multiplications
- When the number of phases increase (b decreases), A is re-broadcasted for each phase, increasing communication



A. Azad, G. Pavlopoulos, C. Ouzounis, N. Kyrpides, A. Buluç; HipMCL: a high-performance parallel implementation of the Markov clustering algorithm for large-scale networks, *Nucleic Acids Research, 2018*

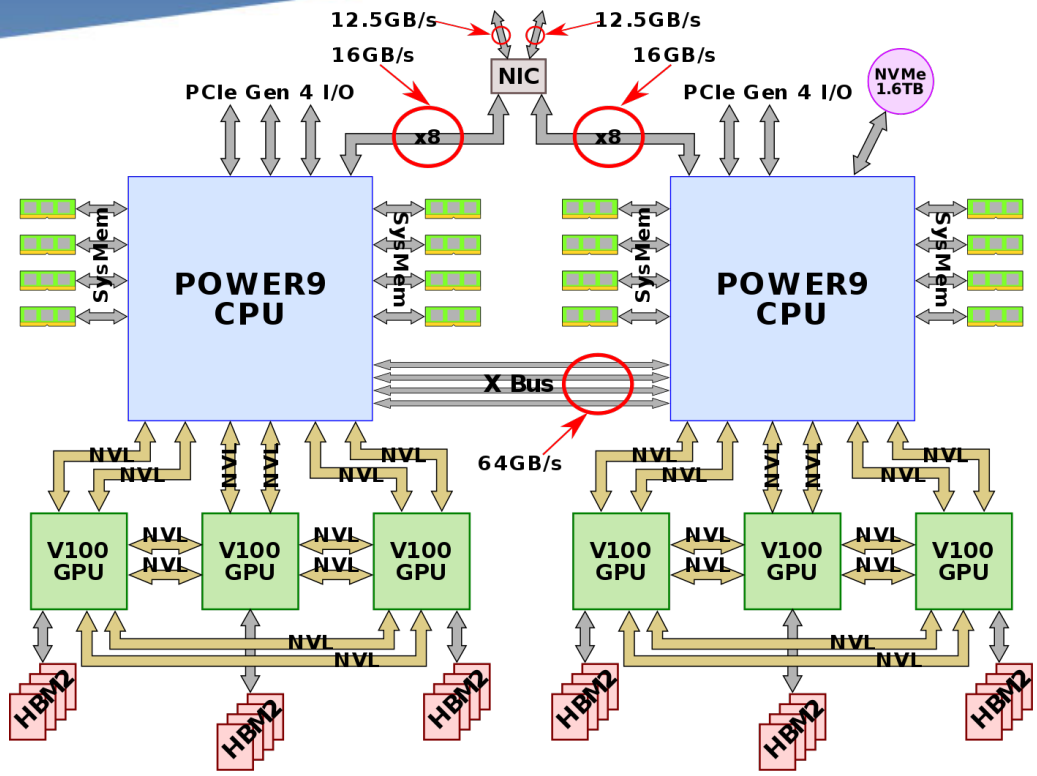
HipMCL on large networks

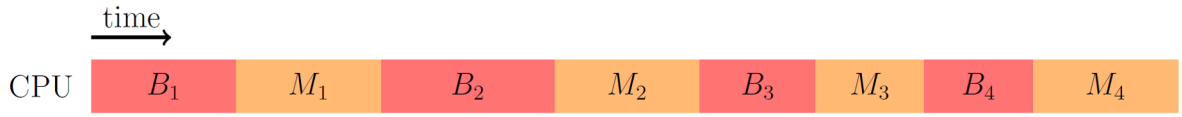
Data	Proteins	Edges	#Clusters	HipMCL time	platform
Isolate-1	47M	7 B	1.6M	1 hr	1024 nodes Edison
Isolate-2	69M	12 B	3.4M	1.66 hr	1024 nodes Edison
Isolate-3	70M	68 B	2.9M	2.41 hr	2048 nodes Cori KNL
MetaClust50	282M	37B	41.5M	3.23 hr	2048 nodes Cori KNL

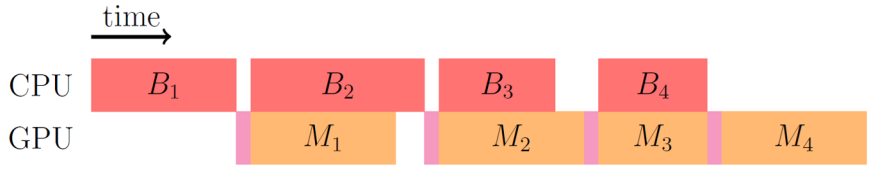
MCL can not cluster these networks

HipMCL on Supercomputers with accelerators

- Recent top supercomputers are all accelerated (e.g. with GPUs)
- This is what a ORNL Summit node looks like
- There are 4608 such nodes in the system
- Challenges: (1) Utilizing all GPUs, (2) hiding the communication



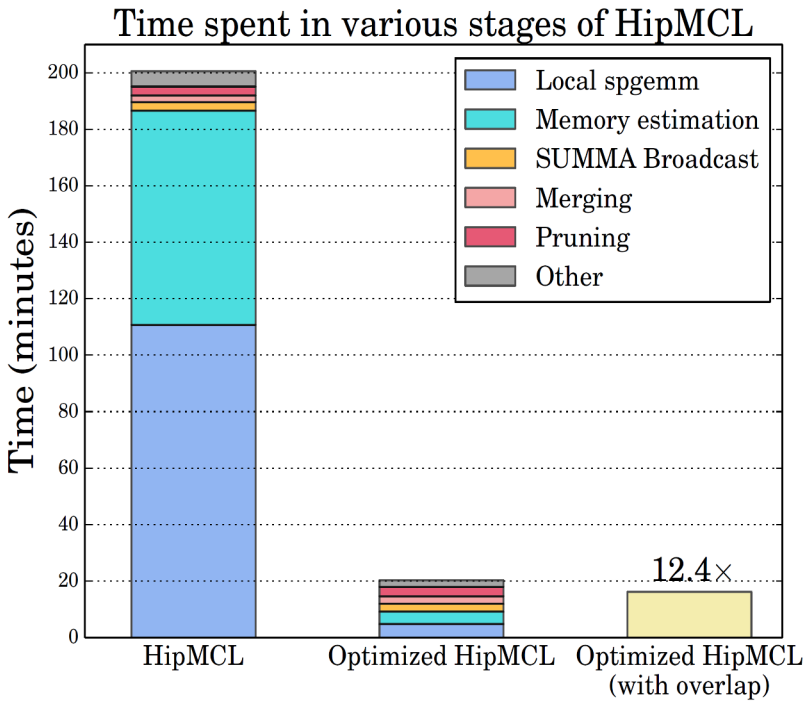


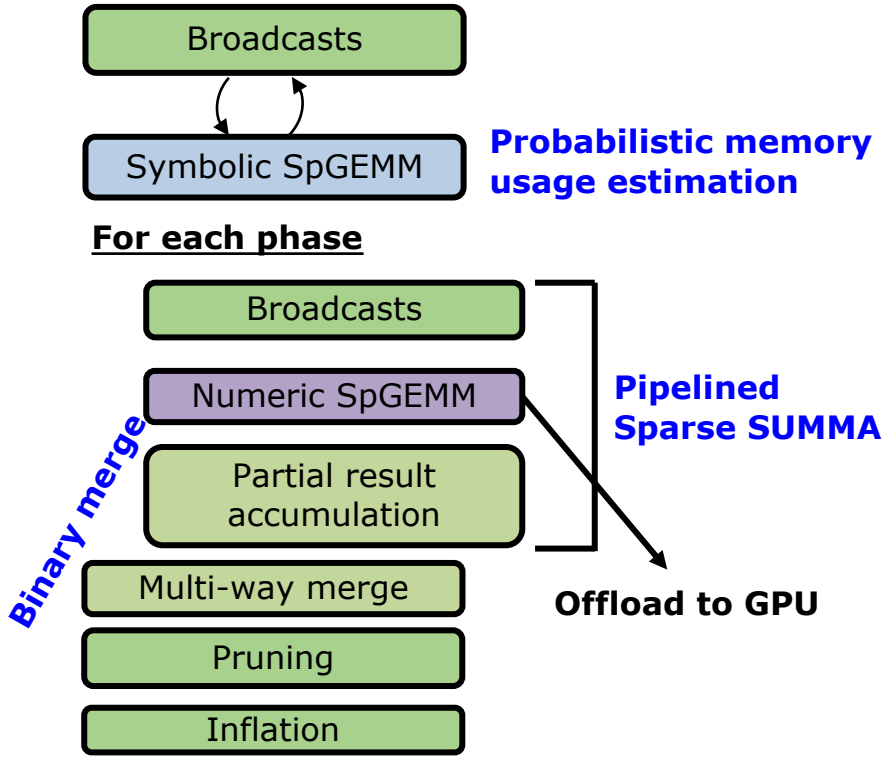


Pipelined Sparse SUMMA

Joint CPU-GPU distributed memory expansion of MCL algorithm

HipMCL on Supercomputers with accelerators





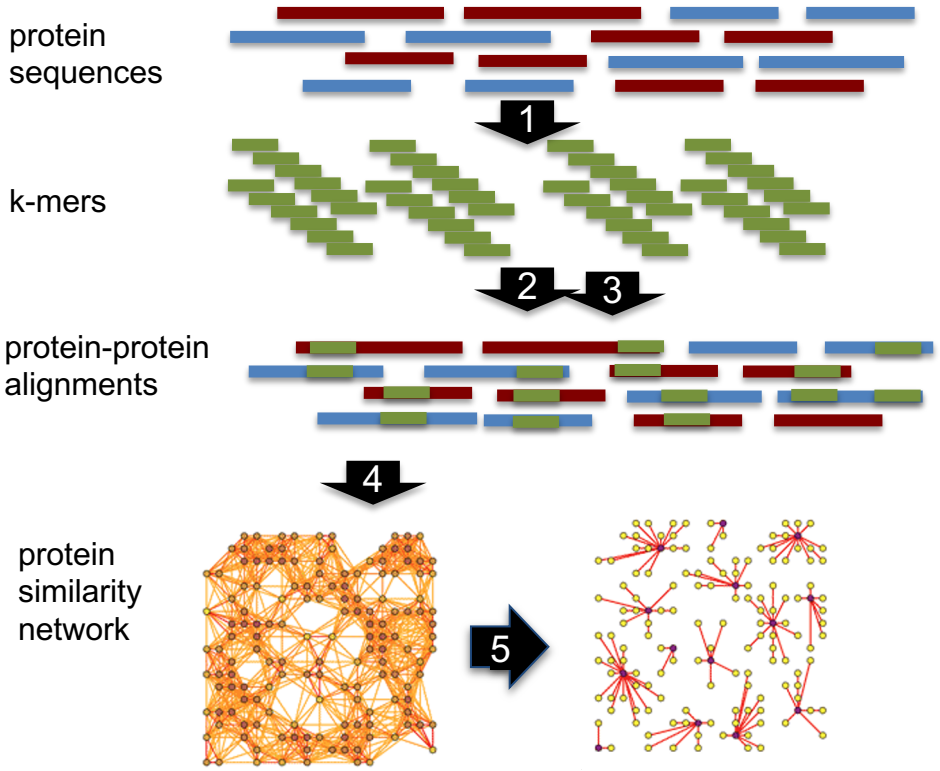
Other changes to HipMCL for the CPU-GPU workflow:

- *Randomized memory estimation algorithm* avoids symbolic phase
- New *eager binary merging* reduces memory footprint
- Integration of a much faster hash-based CPU SpGEMM algorithm

O. Selvitopi, M.T. Hussain, A. Azad, and A. Buluç. Optimizing high performance Markov clustering for pre-exascale architectures. IPDPS, 2020

Recap

PASTIS + HipMCL approach for protein family identification



1) K-mer Analysis
K-mer histogram

2) Sparse matrix building
A: proteins-by-kmers

3) Overlapping via SpGEMM
$C = AA^T$ (or ASA^T)

4) Pairwise alignments
M = filter(C, alignment_score)

5) MCL iteration via SpGEMM
$M^{i+1} = \text{Prune}(M^i M^i)$

SpGEMM: Sparse matrix times sparse matrix

Outline

- Genome (and metagenome) assembly problems
- Indexing with hash tables
- The sparse matrix approach
- Protein family identification
- **Pairwise sequence alignment**
- Indexing with suffix arrays

Pairwise sequence alignment

1. Finding globally best alignment

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

Needleman-Wunsch

match = 1 mismatch = -1 gap = -1

		G	C	A	T	G	C	U
	0	-1	-2	-3	-4	-5	-6	-7
G	-1	1	0	-1	-2	-3	-4	-5
A	-2	0	0	1	0	-1	-2	-3
T	-3	-1	-1	0	2	1	0	-1
T	-4	-2	-2	-1	1	1	0	-1
A	-5	-3	-3	-1	0	0	0	-1
C	-6	-4	-2	-2	-1	-1	1	0
A	-7	-5	-3	-1	-2	-2	0	0

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

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

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

Initialize the scoring matrix

		T	G	T	T	A	C	G	G
	0	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: $S(a_i, b_j) = \begin{cases} +3, & a_i = b_j \\ -3, & a_i \neq b_j \end{cases}$

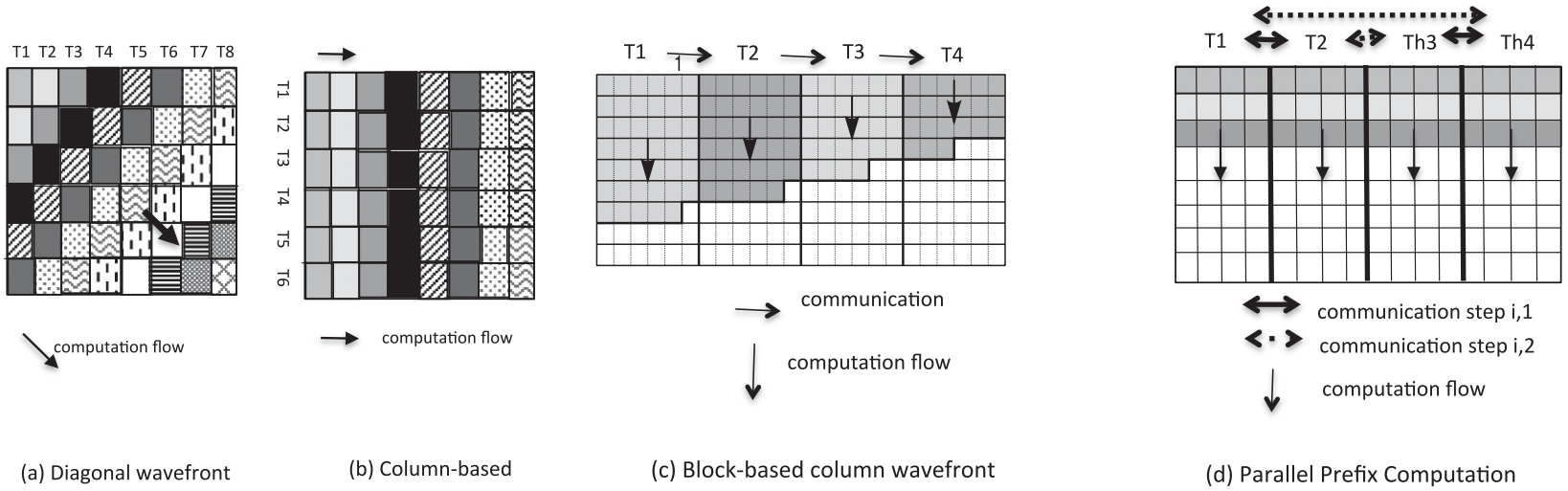
Gap penalty: $W_k = kW_1$, $W_1 = 2$

Smith-Waterman Algorithm
Complexity: O(mn)

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

You are never in debt!

Parallel sequence alignment

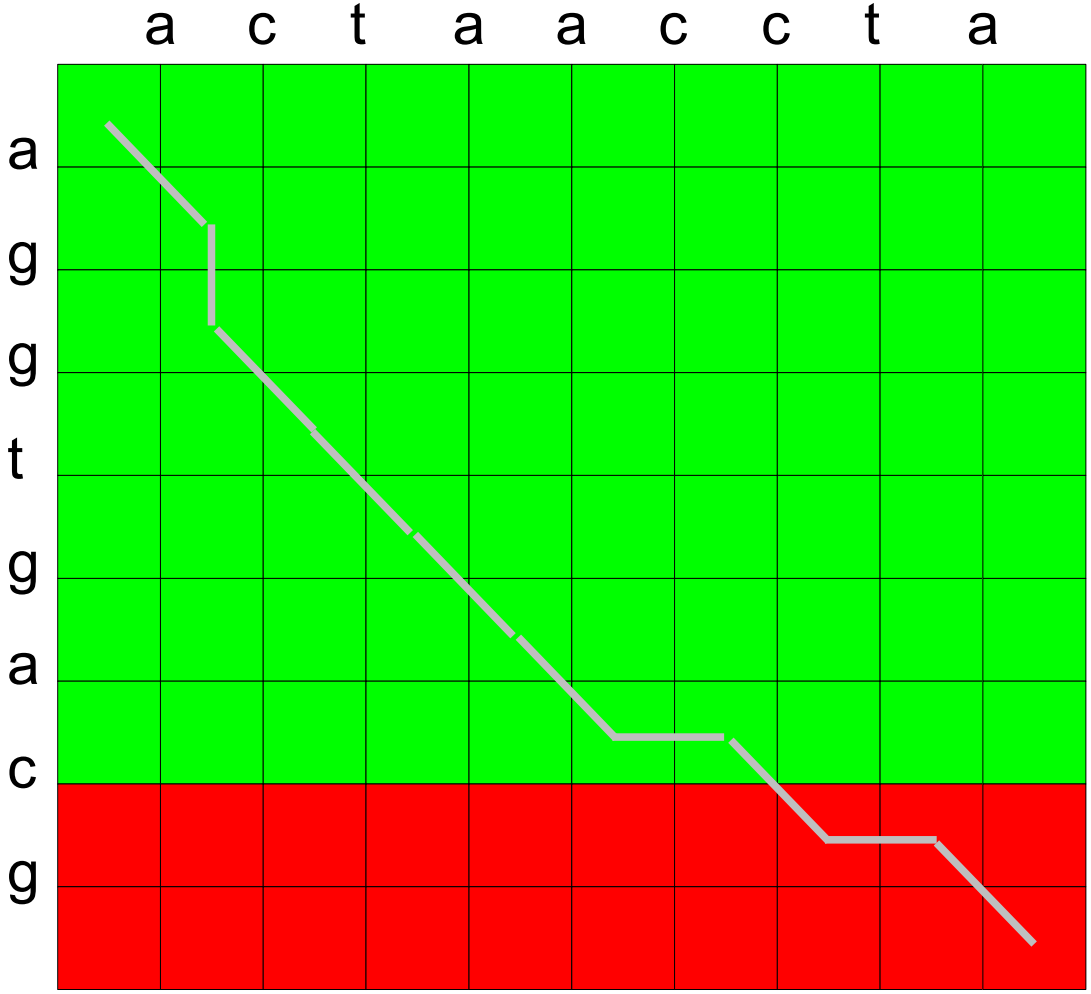


(a) Diagonal wavefront (b) Column-based (c) Block-based column wavefront (d) Parallel Prefix Computation

- 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



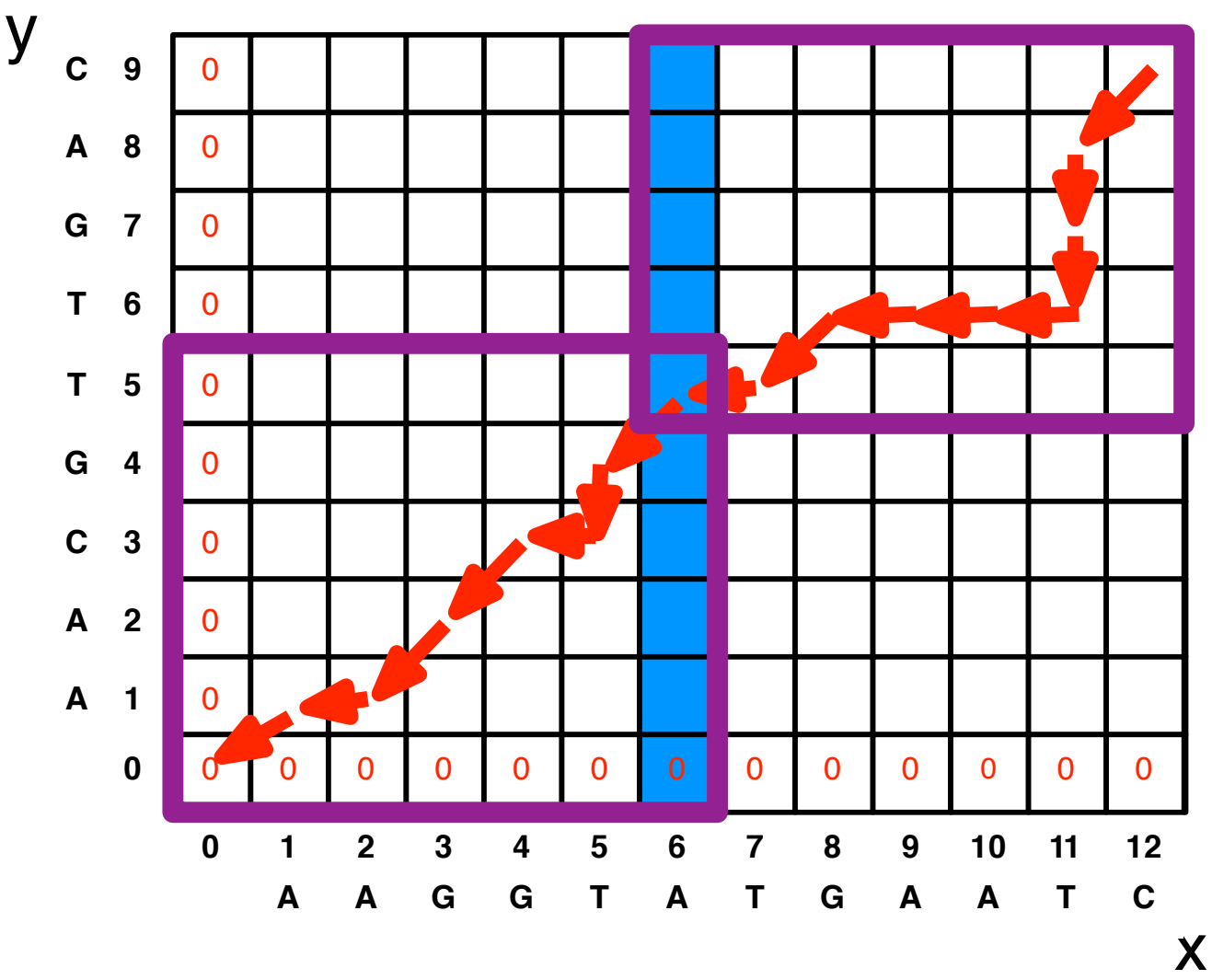
Hirschberg's idea:

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 = ACGTACTG
y = A-GT-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

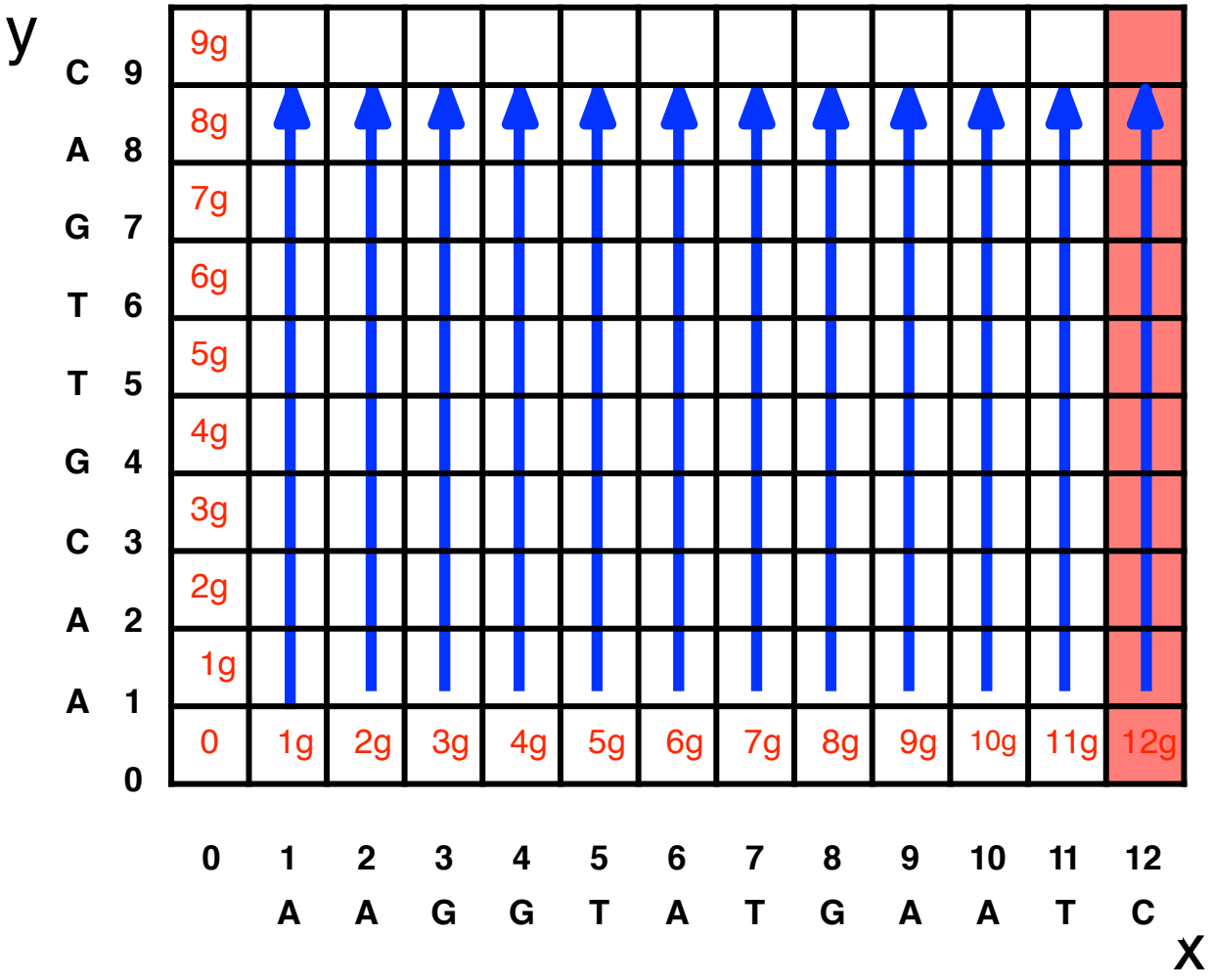
O(n+m) space

find the q that minimizes the cost of the alignment

Problem

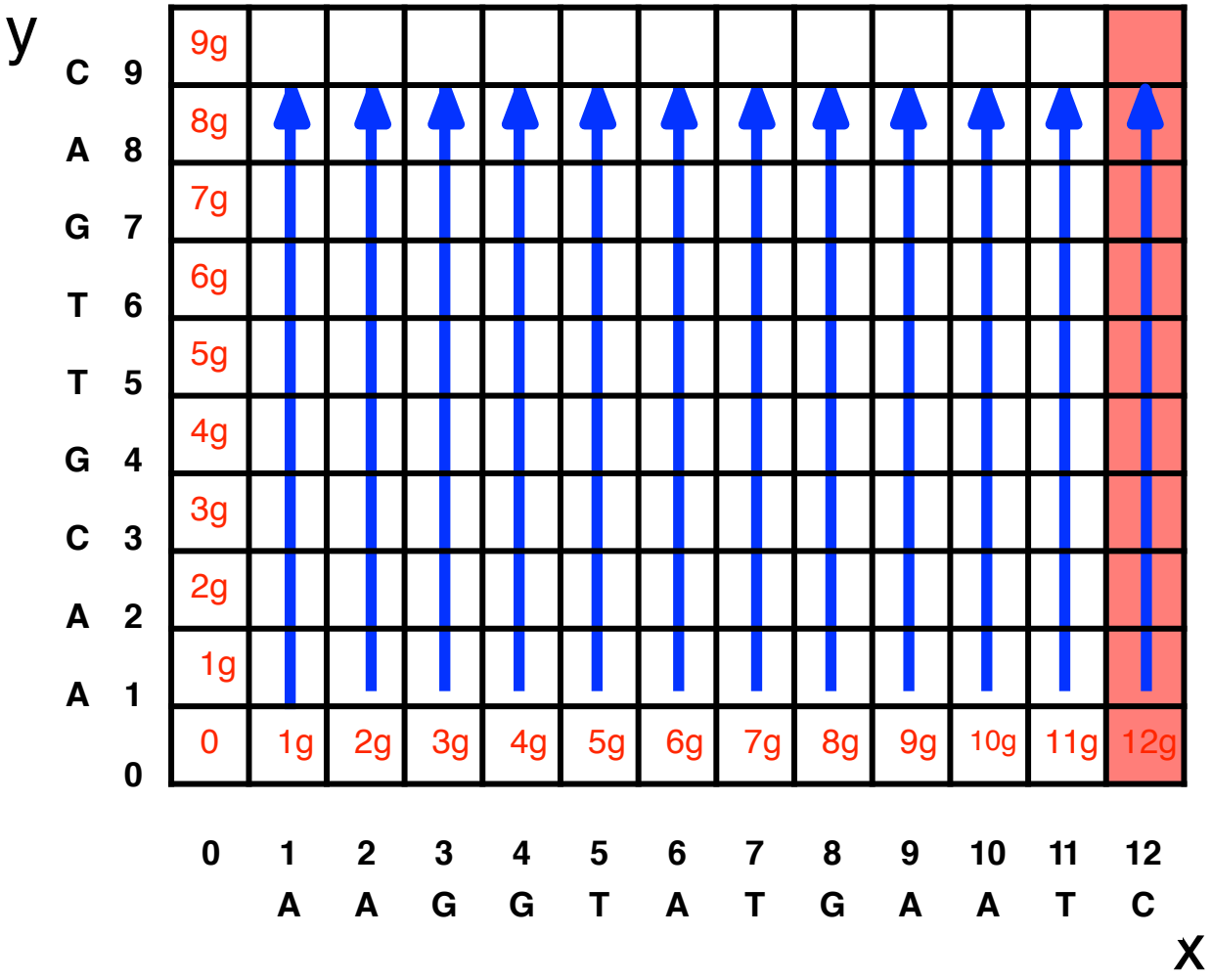
- This works in linear space.
- BUT: not in O(nm) time.
- It's too expensive to solve all those AlignValue problems in the **for** loop.
- Define:
  - **AllYPrefixCosts**(x, i, y) = returns an array of the scores of optimal alignments between x[1..i] and all prefixes of Y.
  - **AllYSuffixCosts**(x,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

Fill in the matrix by columns...



What is this column?

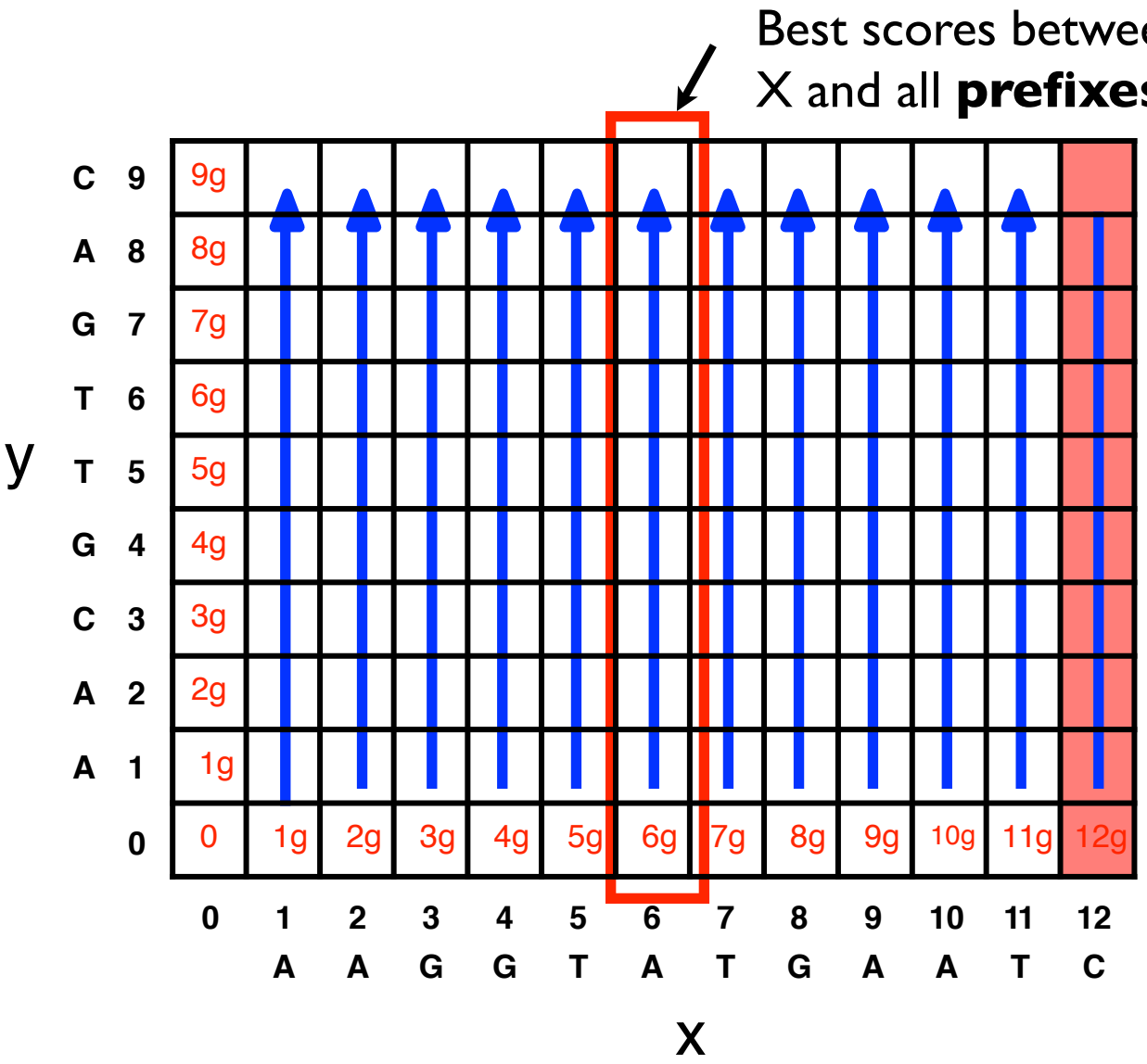
Fill in the matrix by columns...



What is this column?

Best scores between X and all prefixes of Y

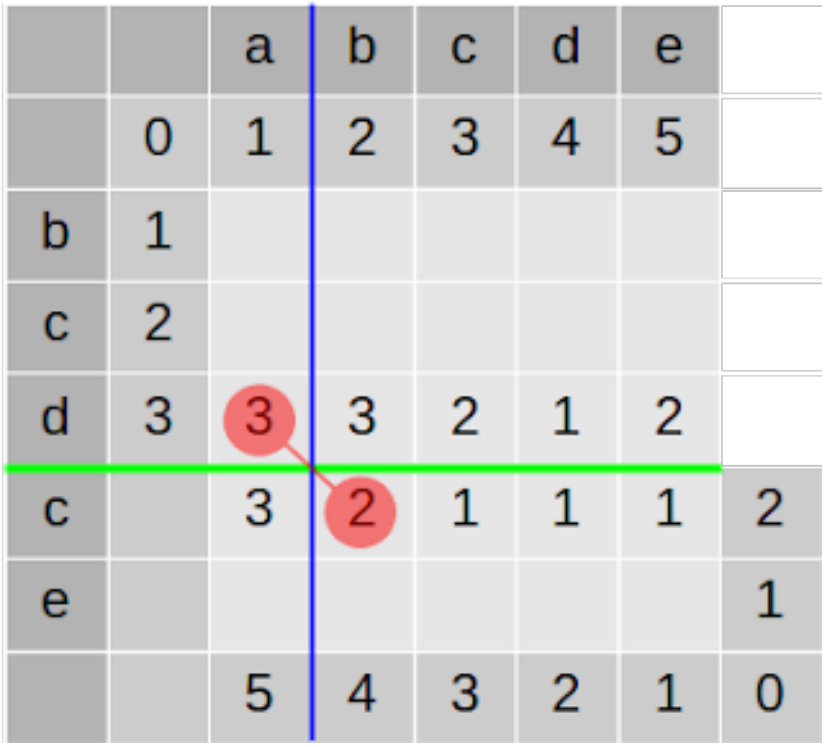
Fill in the matrix by columns...



Computing **scores** for a given column is O(mn) and takes linear space using Needleman-Wunsch

How about the suffixes?

- We find optimal divisions using the score-only version of Needleman-Wunsch
- Then we construct the traceback by recursing along those divisions.



- How to get optimal scores between a suffix of X and all **suffixes** of Y?
- Just run Needleman-Wunsch **backwards** from bottom-right

(this division shown is obviously not optimal by the way)

Space Efficient Alignment

```
   12345678
x = ACGTACTG
y = A-GT-CTG
    q = 3
```

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.

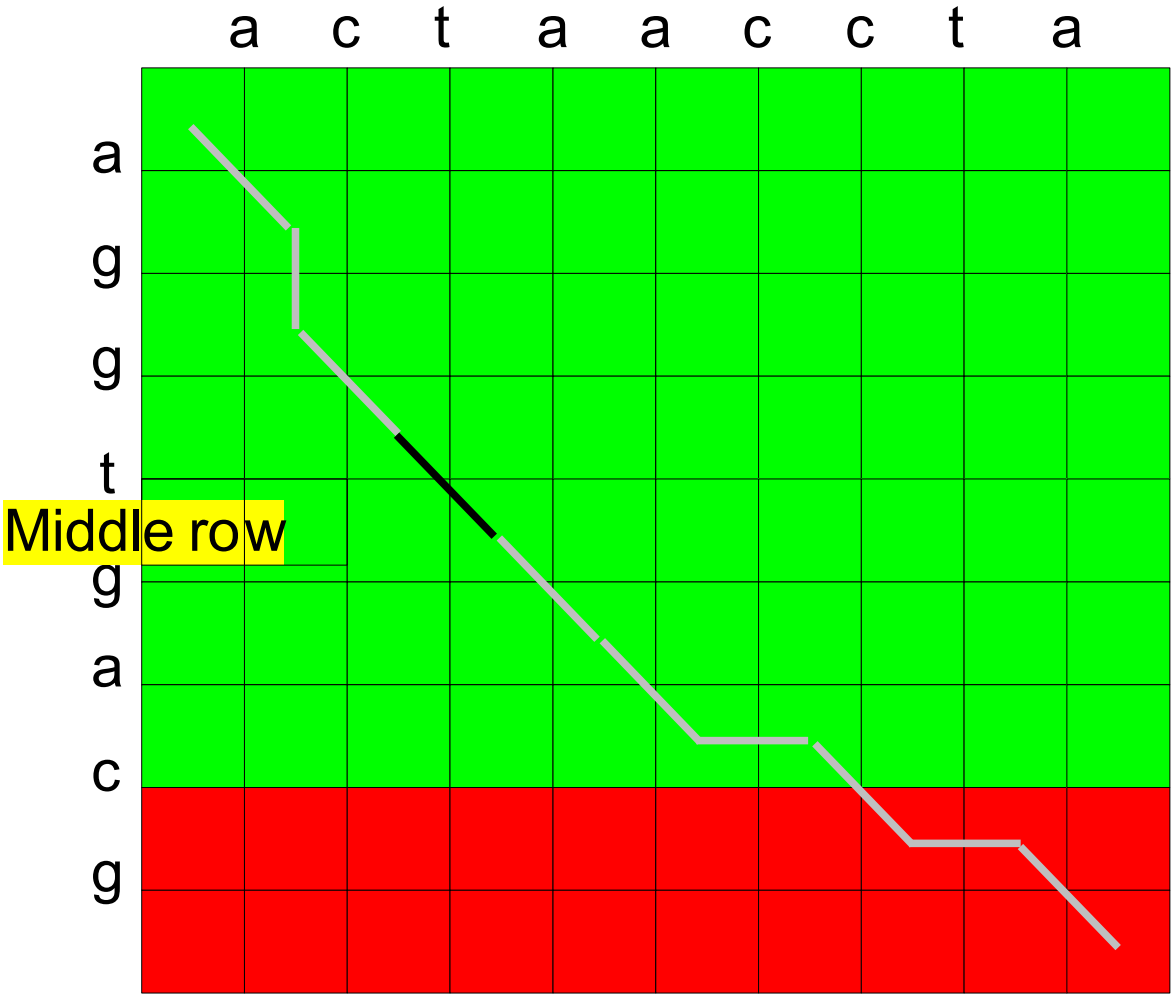
```
ArrowPath := []

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

  YPrefix := AllYPrefixCosts(x, n/2, y)
  YSuffix := AllYSuffixCosts(x, n/2+1, y)

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

  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

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

Linear space – Hirschberg's idea



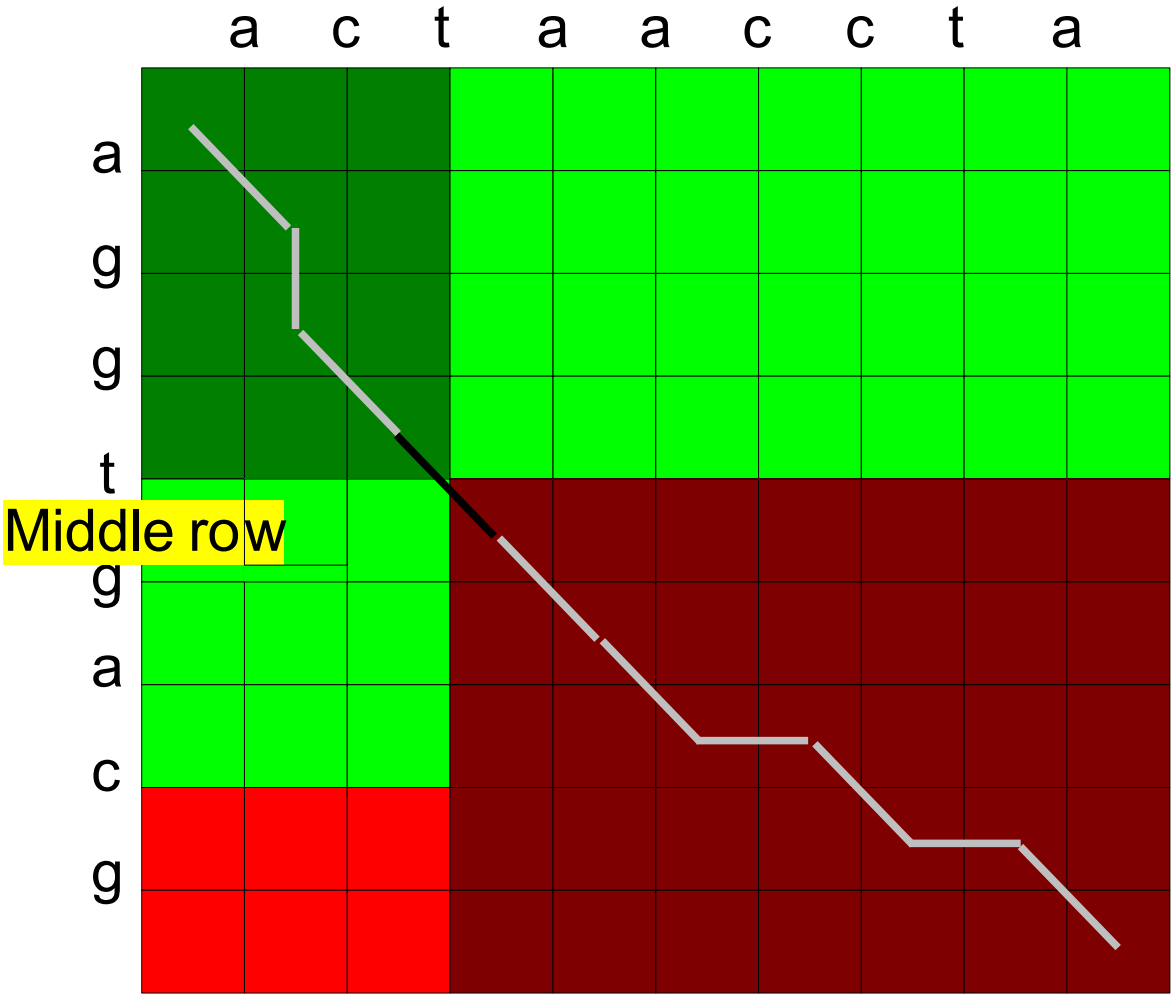
Hirschberg's idea:

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

Time:

nm

Linear space – Hirschberg's idea



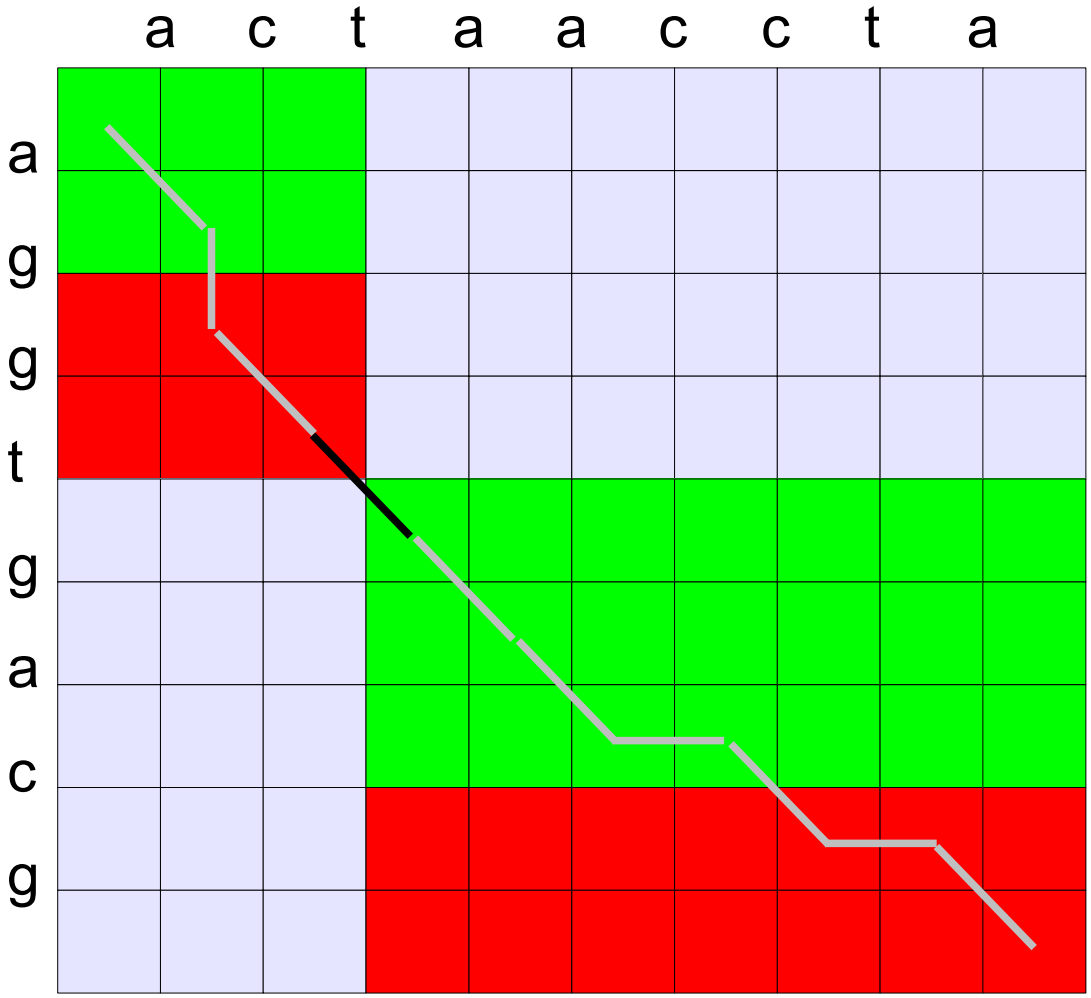
Hirschberg's idea:

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

Time:

nm

Linear space – Hirschberg's idea



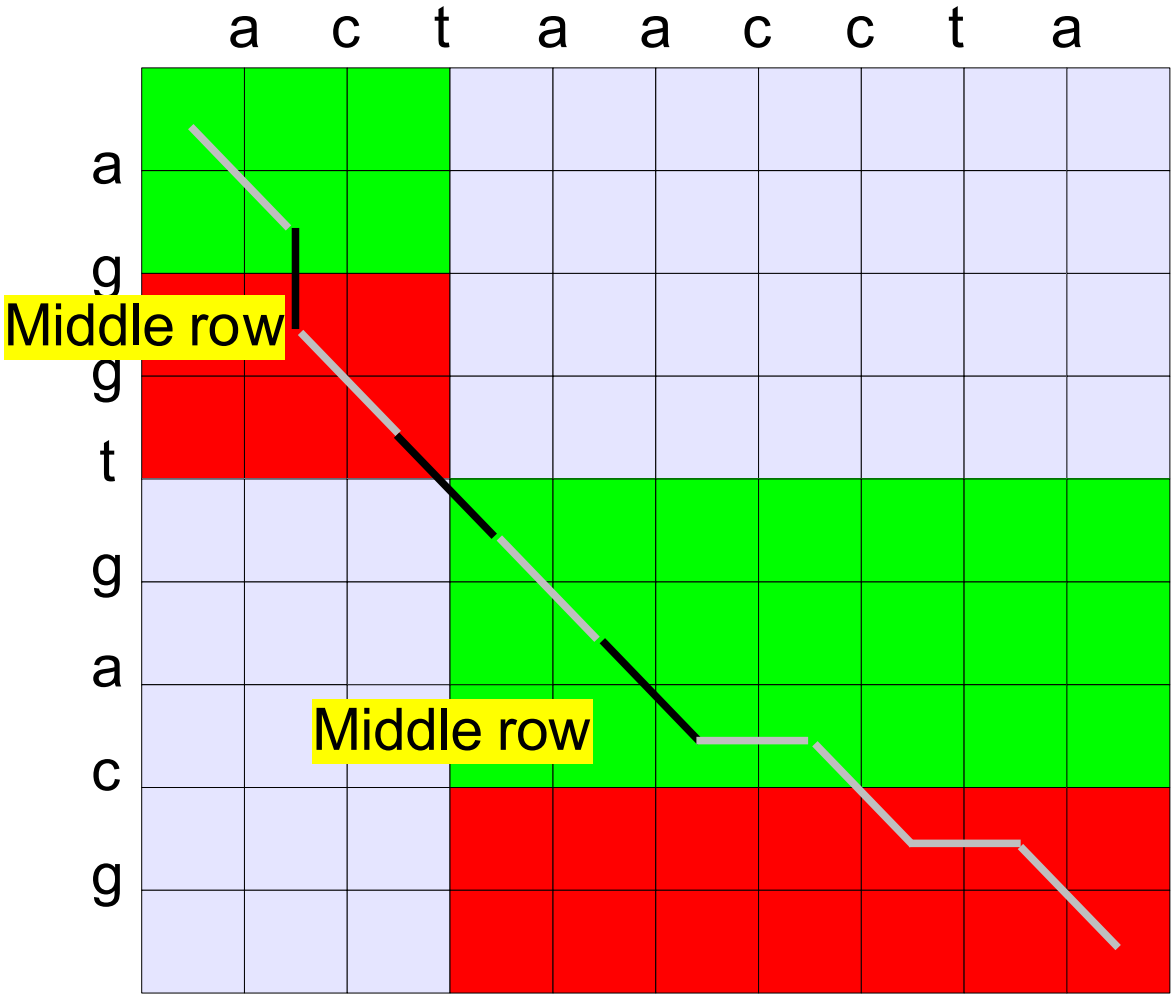
Hirschberg's idea:

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



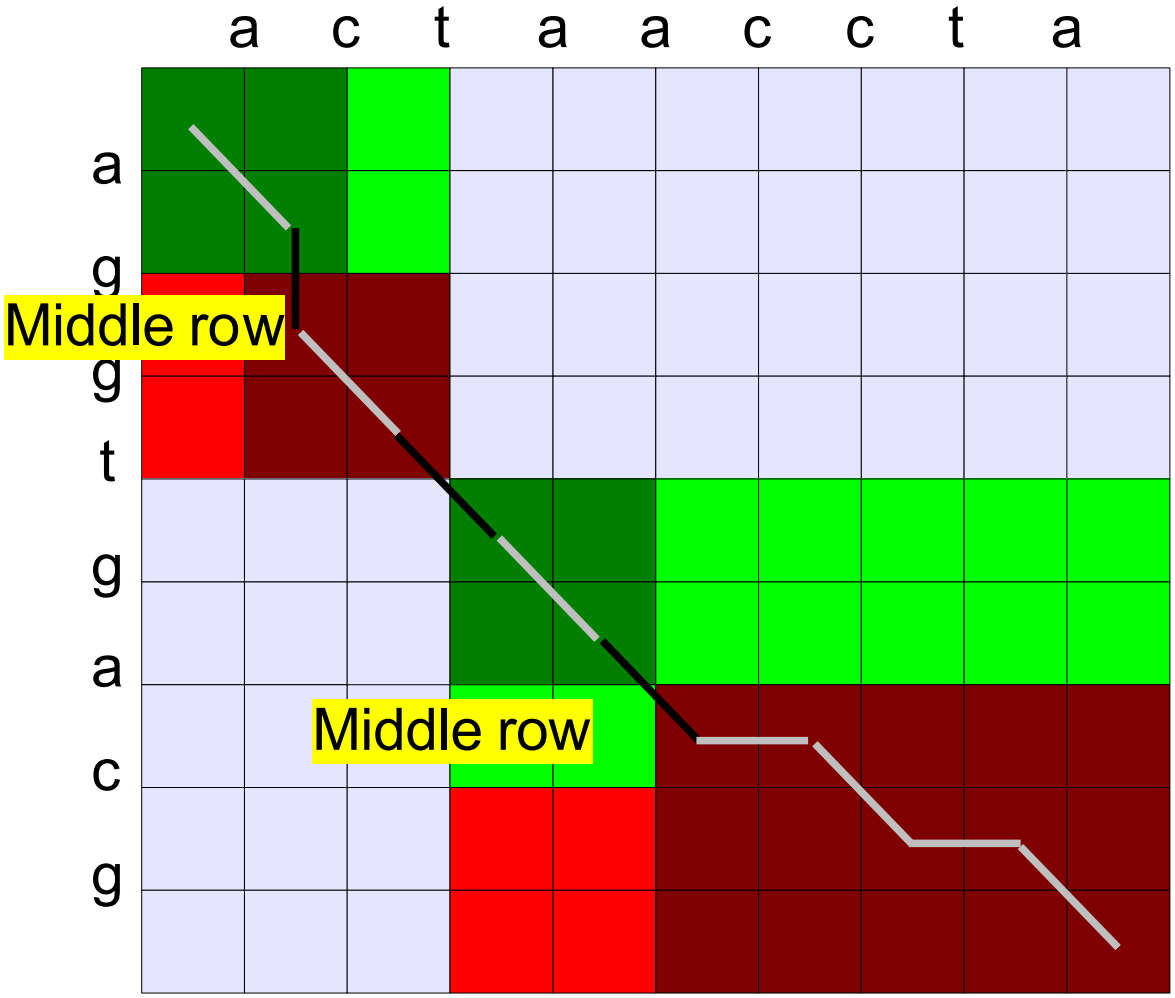
Hirschberg's idea:

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



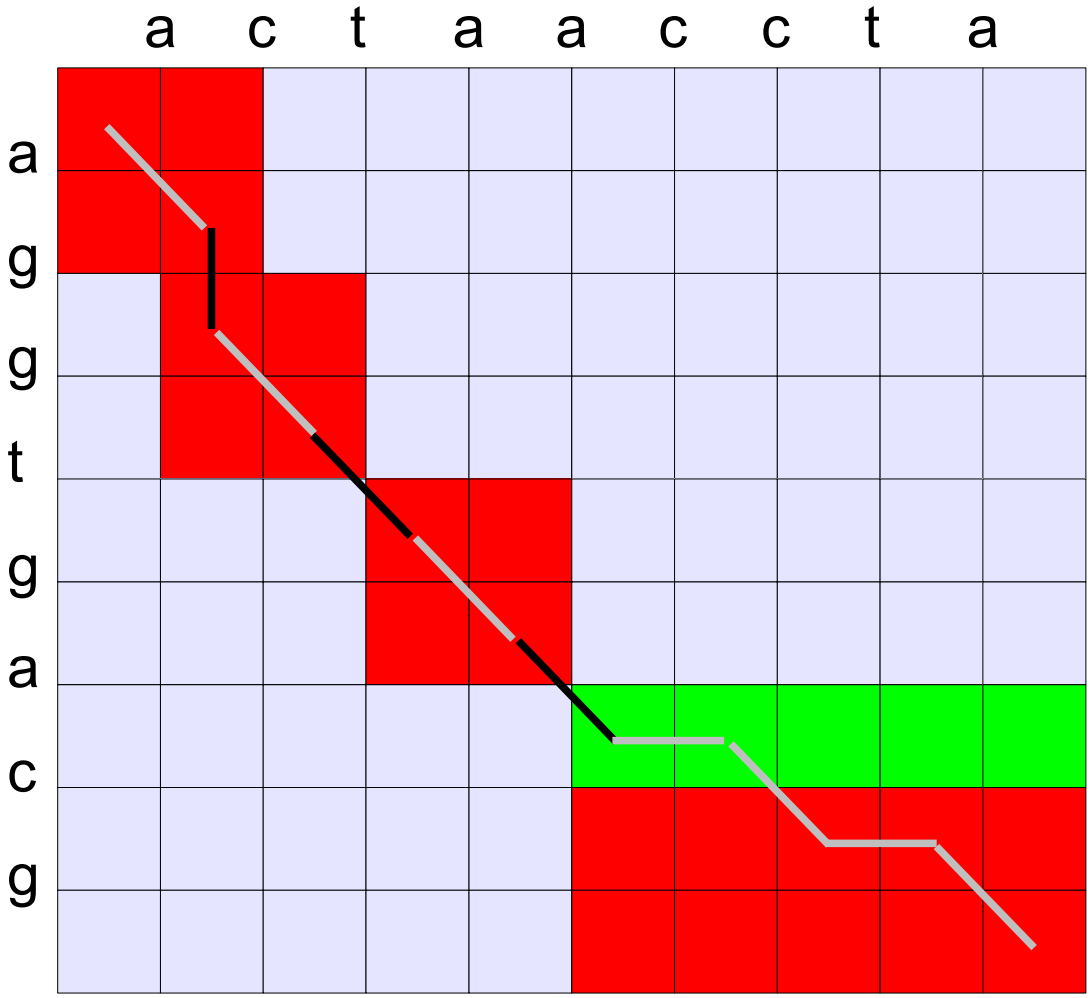
Hirschberg's idea:

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



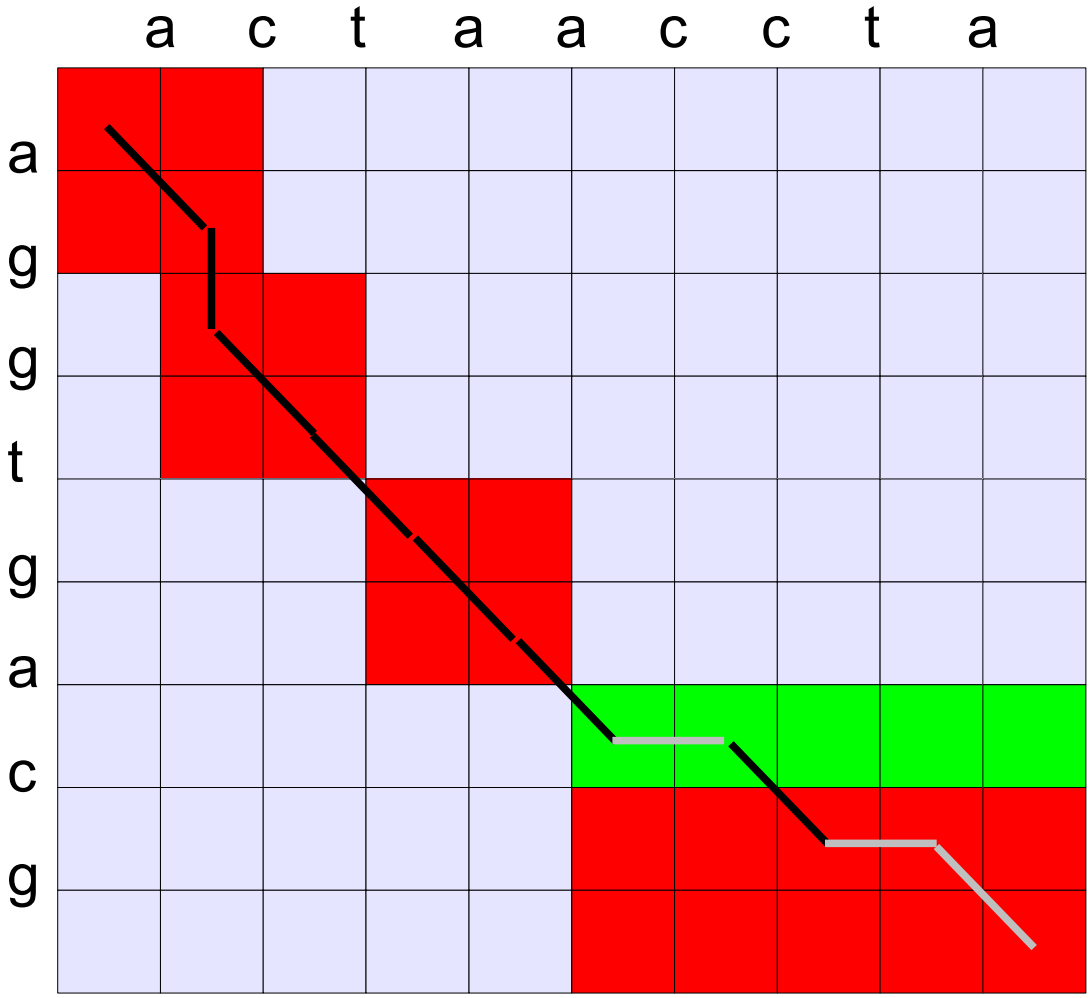
Hirschberg's idea:

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



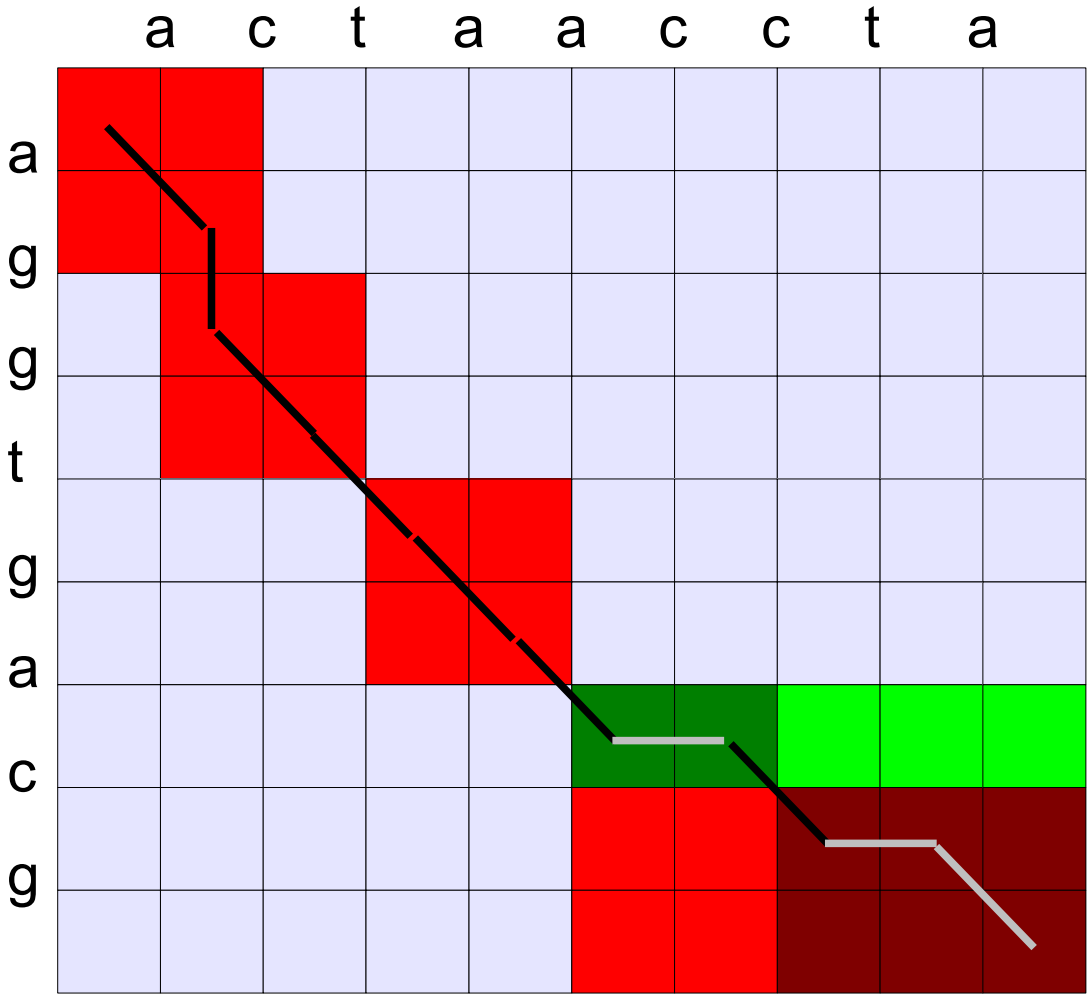
Hirschberg's idea:

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



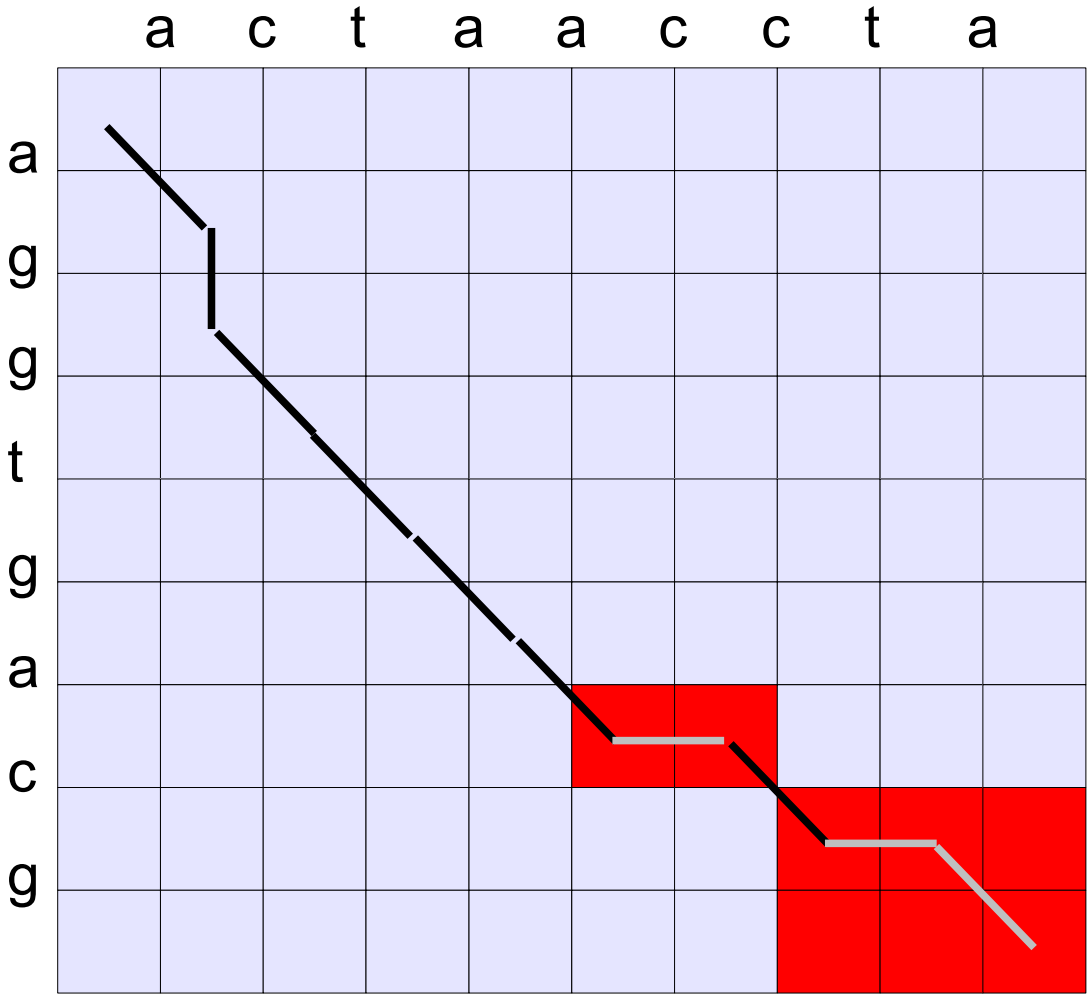
Hirschberg's idea:

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



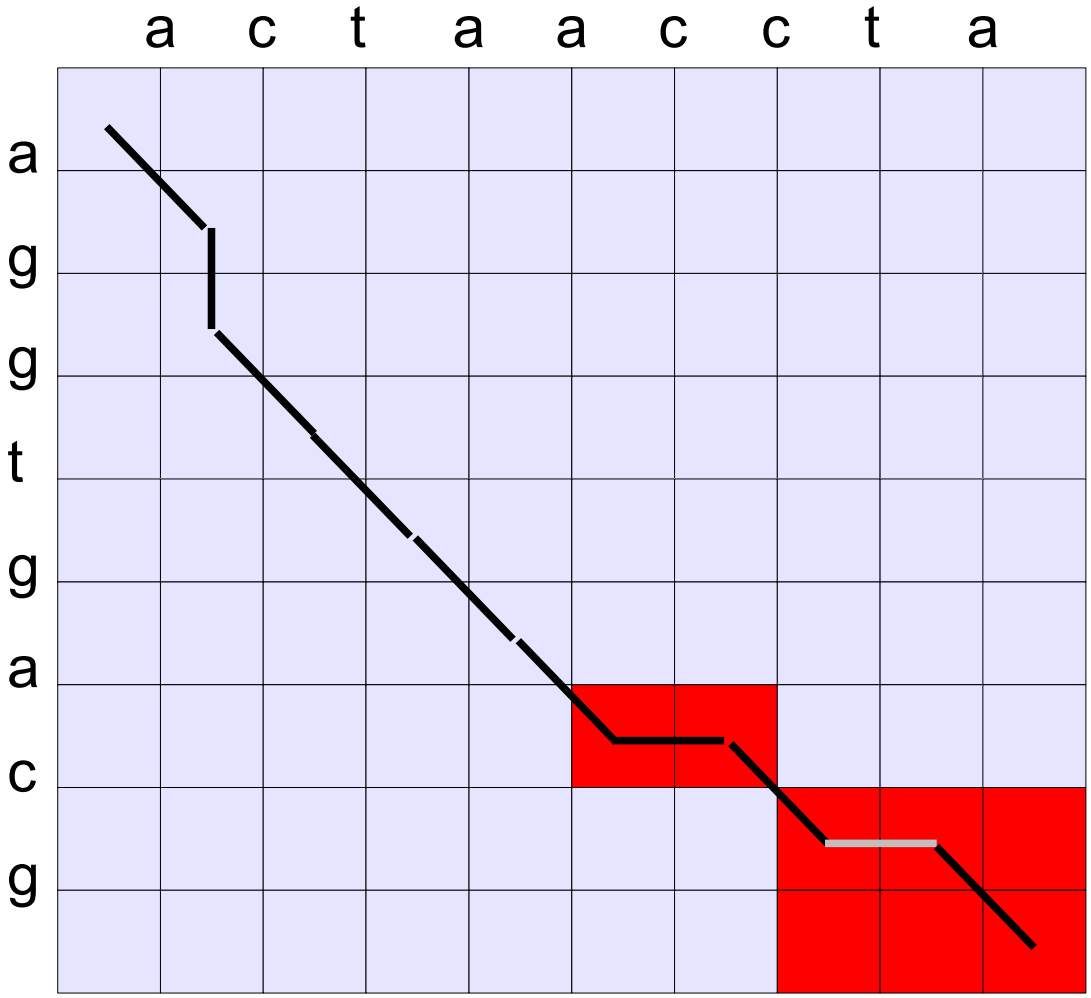
Hirschberg's idea:

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



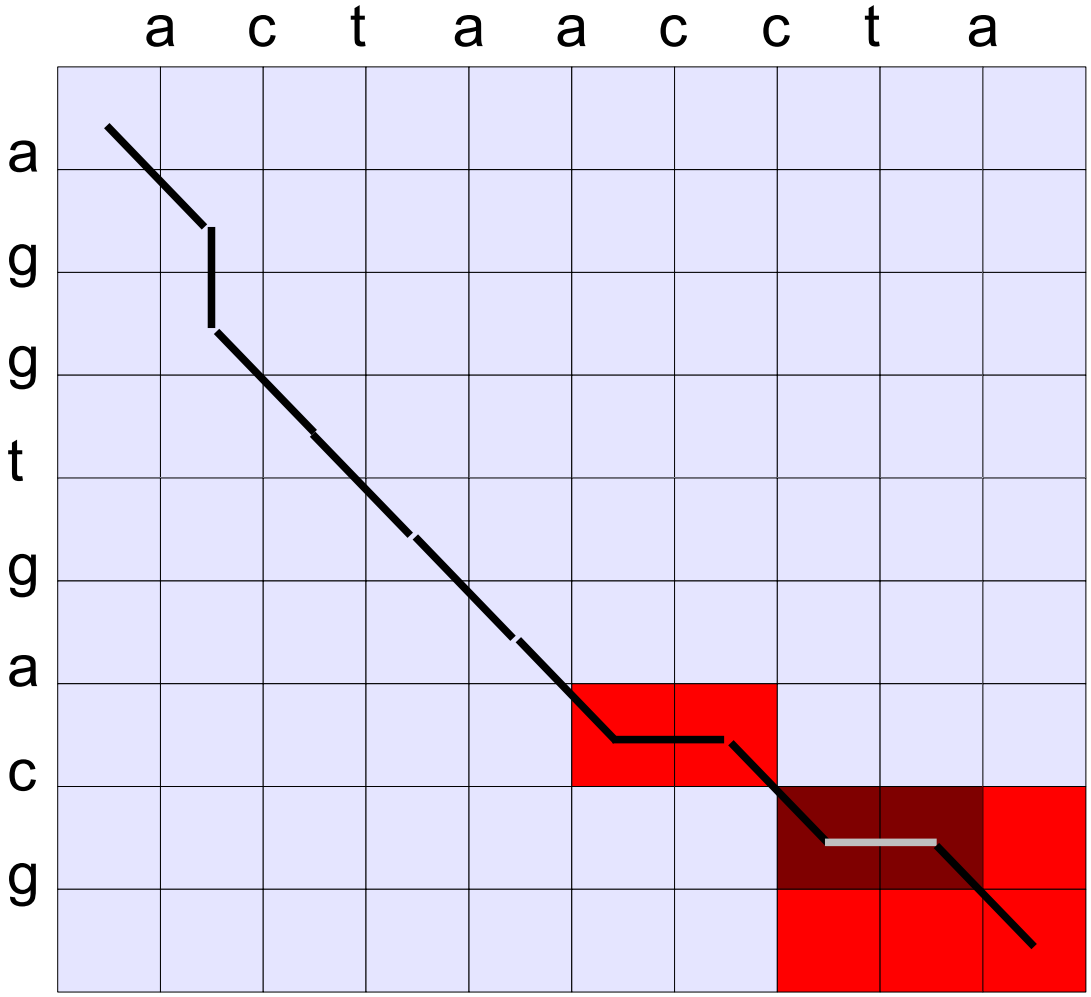
Hirschberg's idea:

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



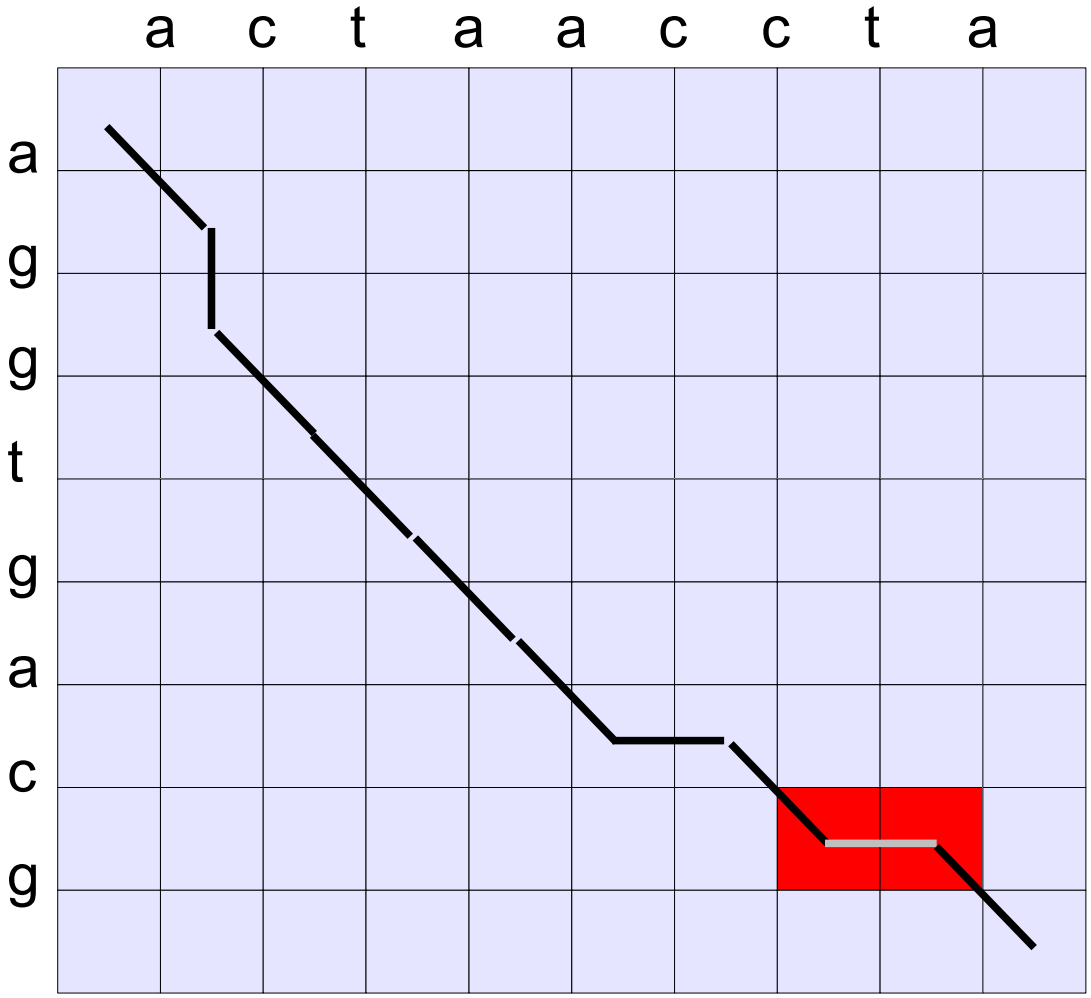
Hirschberg's idea:

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 ...

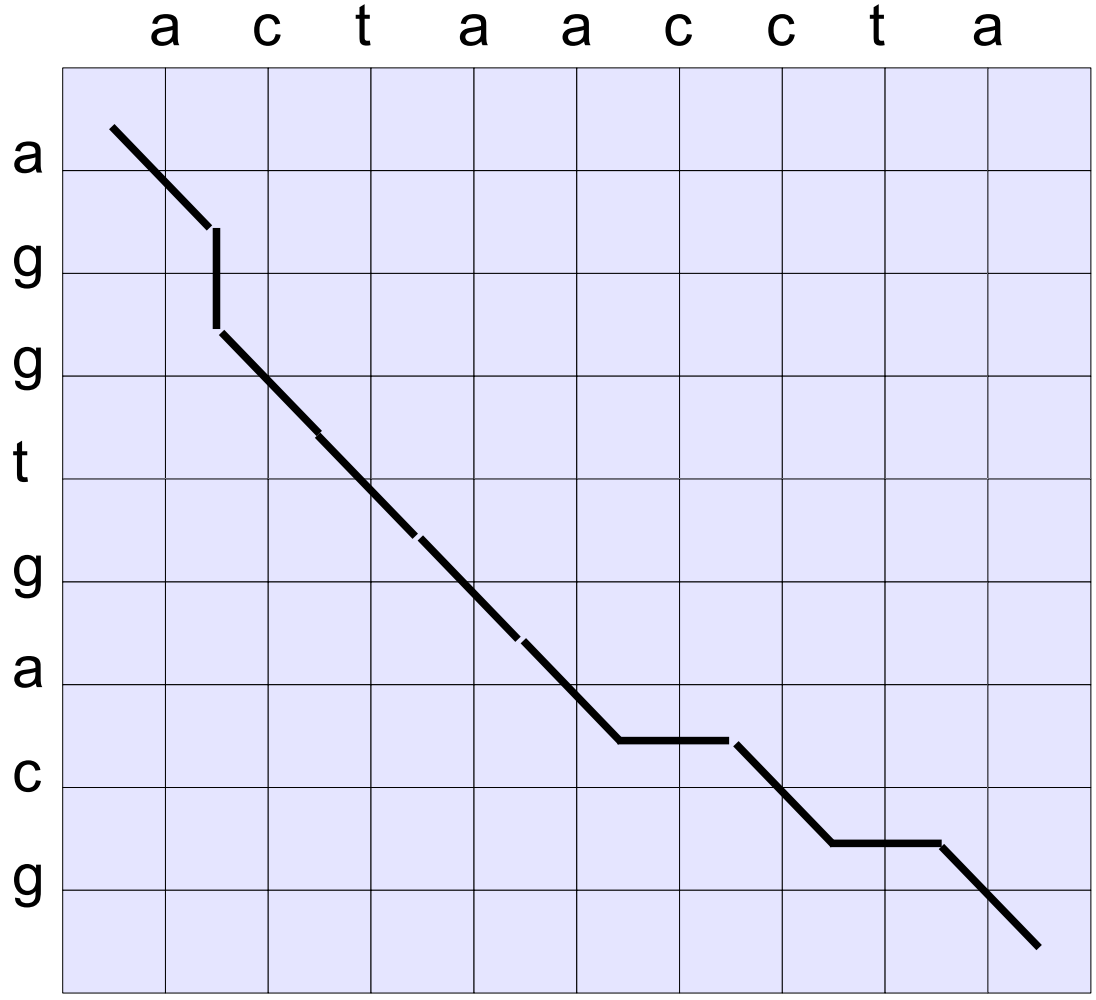
Time:

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

Time analysis (in generel):

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

because $n \le n + (n/2) + (n/4) + \ldots + 2 \le 2n$



Hirschberg's idea:

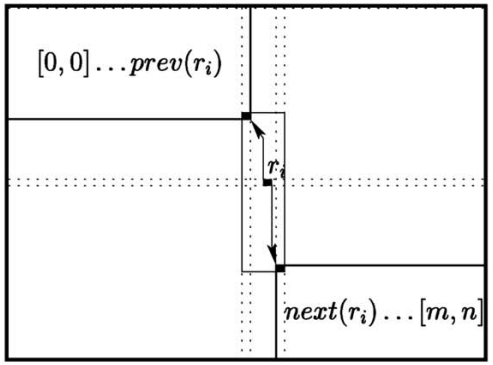
Find “middle edge” (i.e. “middle column”) of an optimal alignment, split into two subproblems and recurse ...

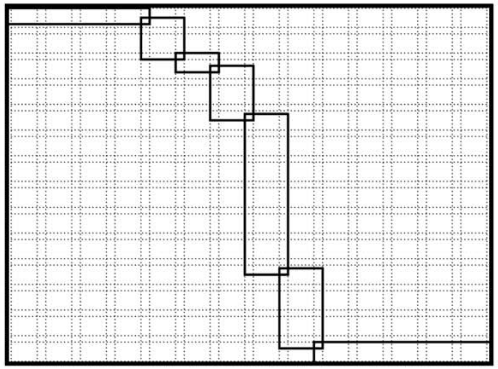
Time:

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

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
- The sparse matrix approach
- Protein family identification
- 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.

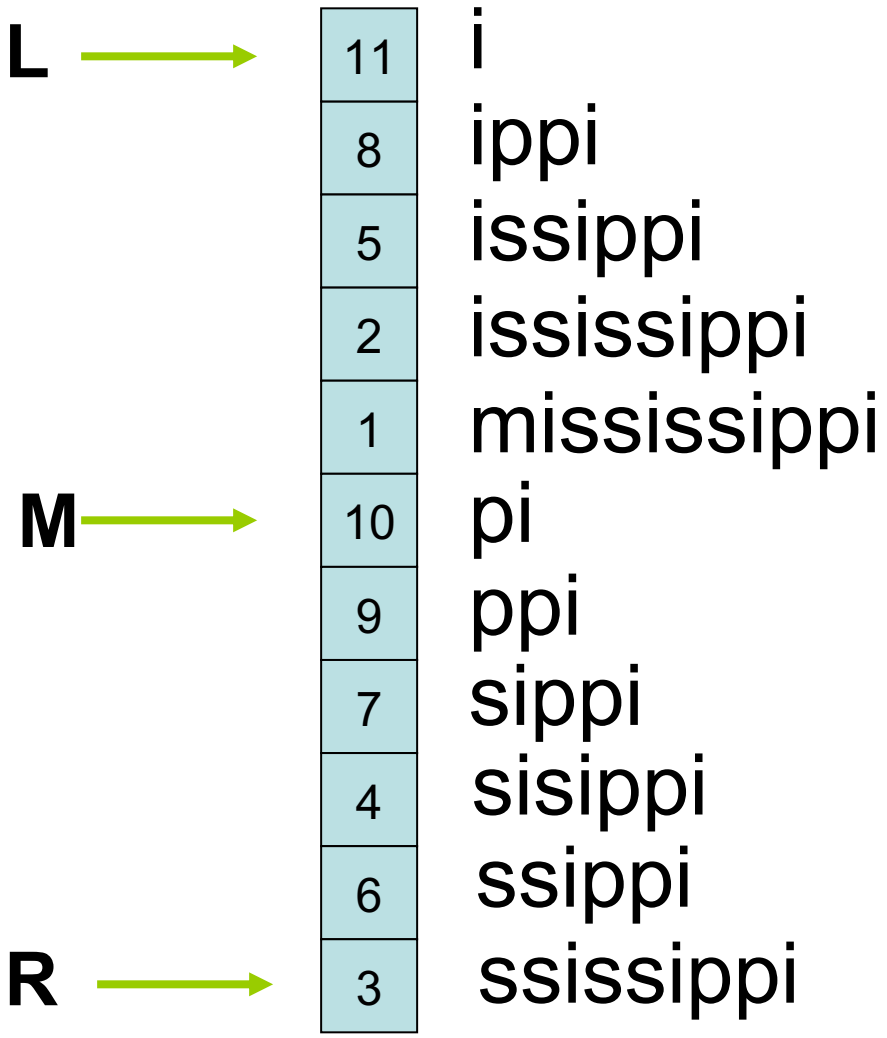
One can construct SAs naively in $O(n^2 \log(n))$ time using string sorting. We can do much better!

Suffix Array Example

Let S = mississippi

Let P = issip

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 mod 3 ≠ 0

1: i s s i s s i p p i
2: s s i s s i p p i
4: i s s i p p i
5: s s i p p i
7: i p p i
8: p p i
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 mod 3 ≠ 0

1: ississippi
2: ssissippi
4: issippi
5: ssippi
7: ippi
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 mod 3 ≠ 0

```
 1: ississippi
 2: ssissippi
 4: issippi
 5: ssippi
 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 mod 3 ≠ 0

```
 1:  ississippi
 2:  ssissippi
 4:  issippi
 5:  ssippi
 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 = "lex-names for i mod 3 =1" # "lex-names for i mod 3 =2" = 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 mod 3 ≠ 0

Step 1 - Compute SA^{12}

Suffixes i mod 3 ≠ 0

```
 1: ississippi
 2: ssissippi
 4: issippi
 5: ssippi
 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
```

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

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

u = "lex-names for i mod 3 =1" # "lex-names for i mod 3 =2" = 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 mod 3 ≠ 0

Step 1 - Why ...

u = 3 3 2 1 # 5 5 4

iss iss ipp i$$ ssi ssi ppi

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 =	3	3	2	1	#	5	5	4
	iss	iss	ipp	i$$		ssi	ssi	ppi

Suffixes of *u*

0:	3321#554	issisippi$$#ssissippi
1:	321#554	issippi$$#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 ...

u = 3 3 2 1 # 5 5 4

iss iss ipp i$$ ssi ssi ppi

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

Since the special character #is unique and occurs in unique positions,

we can sort these strings by sorting the prefixes (of ***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$$ < iss

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

```
 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

				Suffix array	
0:	3321#554	ississippi$$#ssissippi	1	3	10
1:	321#554	issippi$$#ssissippi	4	2	7
2:	21#554	ippi$$#ssissippi	7	1	4
3:	1#554	i$$#ssissippi	10	0	1
4:	554	ssissippi	2	6	8
5:	54	ssippi	5	5	5
6:	4	ppi	8	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 mod 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* mod 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 mod 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 know c.f. 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 2 - Compute SA^0

Suffixes i mod 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	m s[1..]
9:	p	p s[10..]
6:	s	s s[7..]
3:	s	s s[4..]

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

[i+1..] is know c.f. 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 2 - Compute SA^0

Suffixes i mod 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}

... $s[i..]$ is know c.f. 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 know c.f. SA^{12}...

0	9	6	3

10	7	4	1	8	5	2

Example

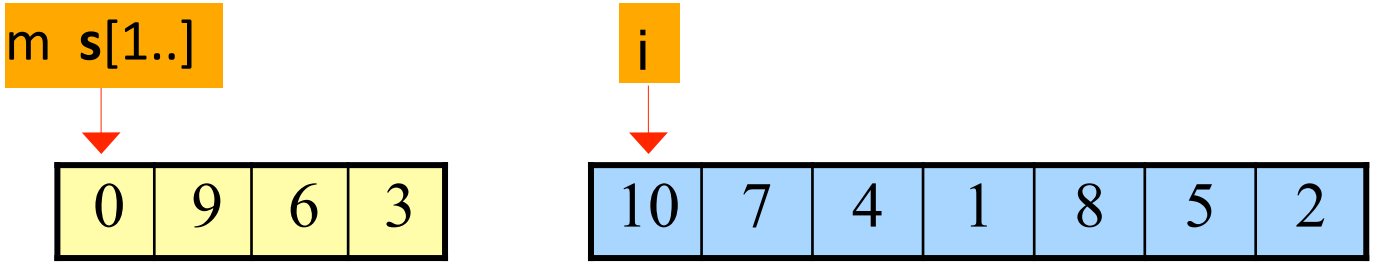
Determine order of suffix $s[1..]$ and $s[4..]$

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

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

$s[5..] < s[2..]$, 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

5: s s i 5

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

Skew Algorithm in Parallel (pDC3)

Function $pDC3(T)$

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

Parallel sort — *sort S by the first component*

Prefix sum — $P := name(S)$

if *the names in P are not unique* **then**

AlltoAllV — *permute the $(r, i) \in P$ such that they are sorted by $(i \bmod 3, i \operatorname{div} 3)$*

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

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

AlltoAllV — *permute P such that it is sorted by the second component*

Data parallel splitting

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

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

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

Parallel sort

$S := $ *sort $S_0 \cup S_1 \cup S_2$ using comparison function:*

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

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

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

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

return $\langle$*last component of s : $s \in S$*$\rangle$

pDC3 Complexity

$$T_{pDC3} = T_{parsort}(n,p) + T_{alltoall}(n/p,p) + f(p)\log(n)$$

Bounds hold as long as

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

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