Parallel de novo Assembly of Complex (Meta) Genomes via HipMer

Aydın Buluç
Computational Research Division, LBNL

May 23, 2016
Invited Talk at HiCOMB 2016

Outline and Acknowledgments

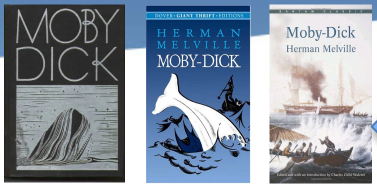
Joint work (alphabetical) with Chaitanya Aluru, Jarrod Chapman, Rob Egan, Evangelos Georganas, Steven Hofmeyr, Lenny Oliker, Dan Rokhsar, Kathy Yelick

1. **Parallel De Bruijn Graph Construction and Traversal for de novo Genome Assembly**, SC'14
2. **A whole-genome shotgun approach for assembling and anchoring the hexaploid bread wheat genome**. Genome Biology, 16(26), 2015.
3. **meraligner: A fully parallel sequence aligner**, IPDPS'15
4. **HipMer: An Extreme-Scale De Novo Genome Assembler**, SC'15

Work is funded by U.S. DEPARTMENT OF ENERGY | Office of Science

De novo Genome Assembly

1. Three copies of the same novel.



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.
```

3. A few strips of characters from one page.

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

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

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
```

1. Three copies of the same DNA.

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

```
ACCGTAGCAAAACCGGGTAGTCATACTACTACGTACTCATCT
```

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 reads

5. Reconstruct original DNA sequence from the read set.

ACCGTAGCAAAACCGGGTAGTCATACTACTACGTACTCATCT

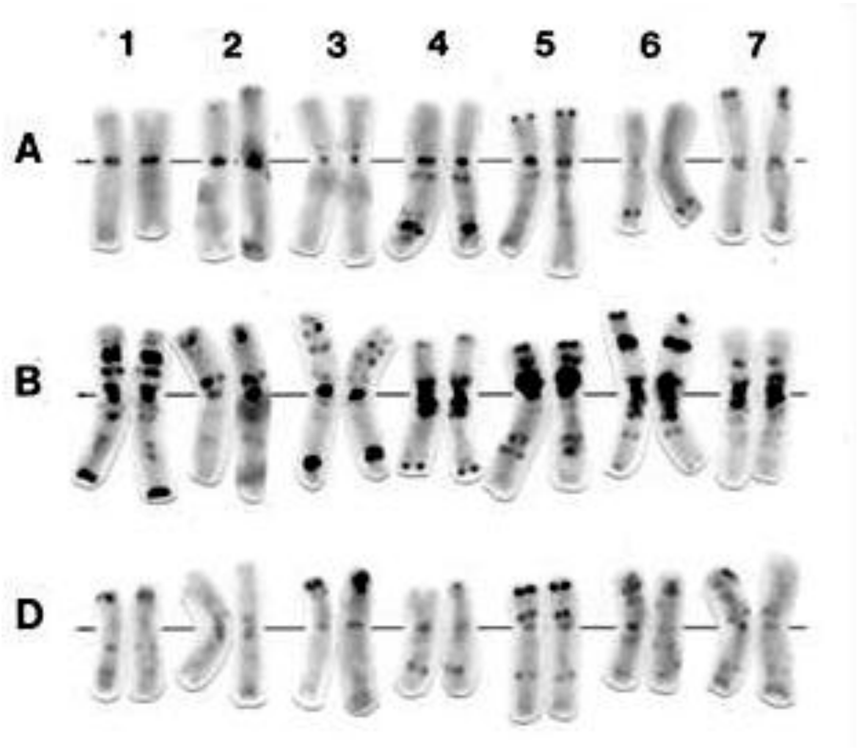
```
ACCGTAGCAA
GTAGTCATACT
AAACCGGGTA
CTACTACGTAC
CGTACTCATCT
```

De novo Genome Assembly is hard

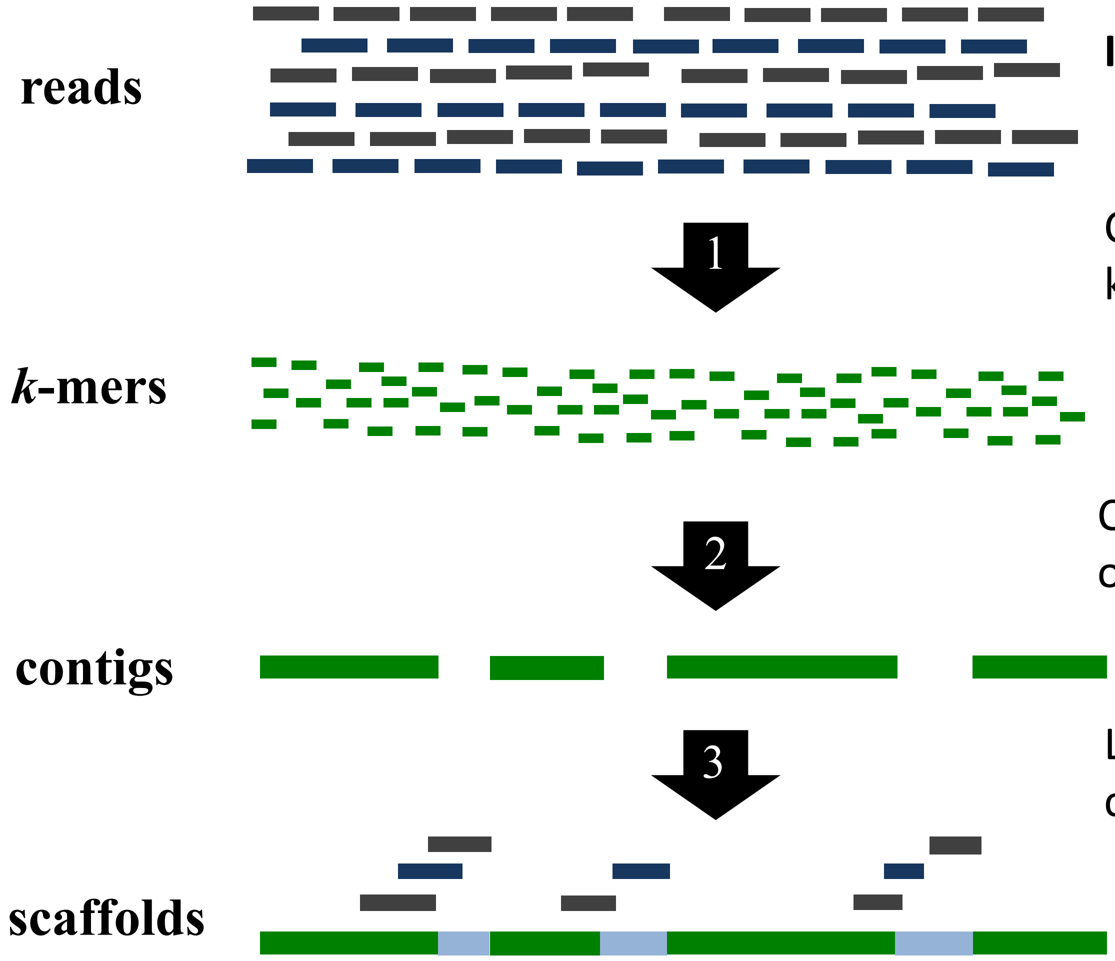
- There is no genome reference!
  - In principle we want to reconstruct unknown genome sequence.
- Reads are significantly shorter than whole genome.
  - Reads consist of 20 to 30K bases
  - Genomes vary in length and complexity – up to 30G bases
- Reads include **errors**.
- Genomes have repetitive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



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)

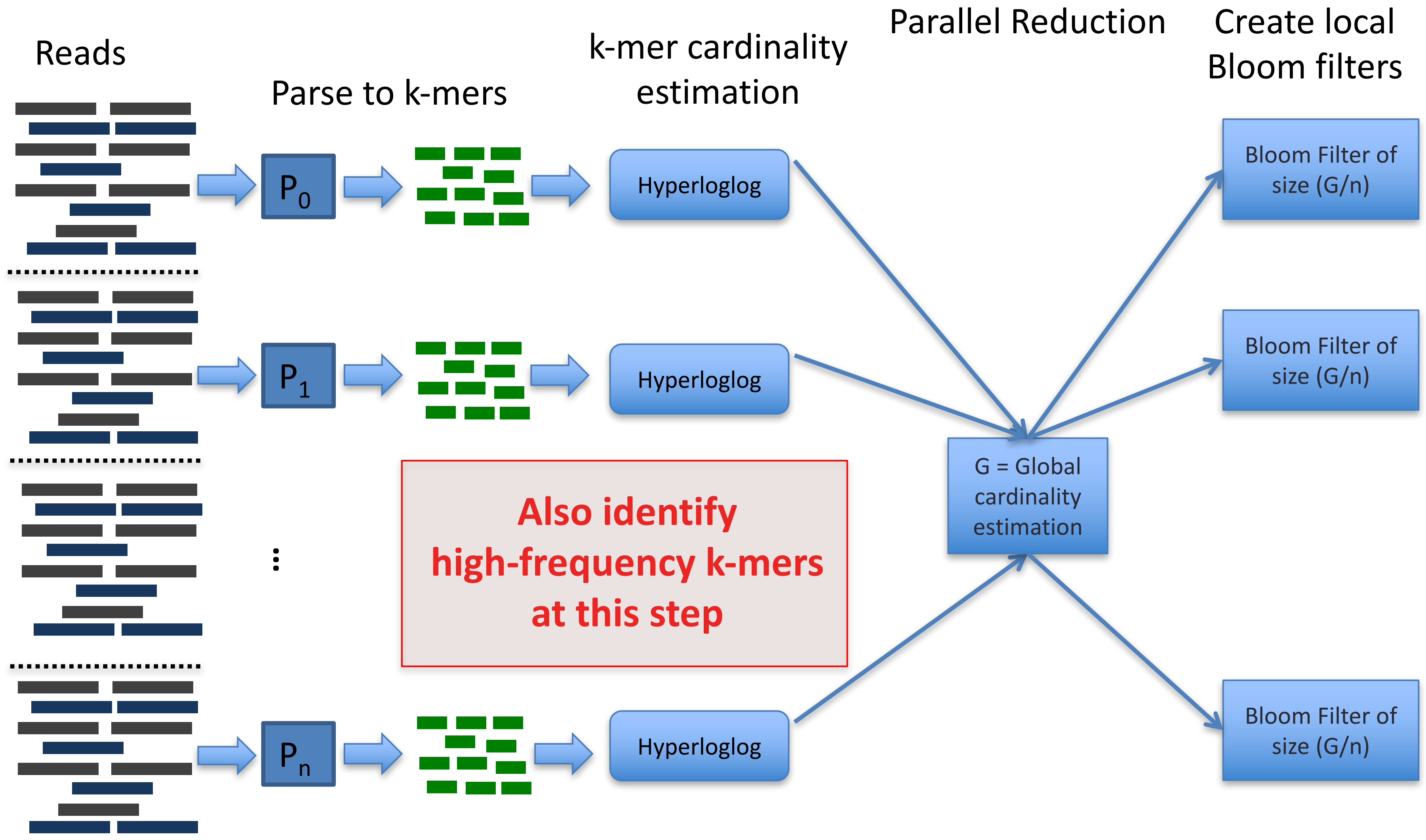
Leverage read information to link contigs and generate scaffolds.

Compute and I/O intensive

Meraculous parallelization with UPC

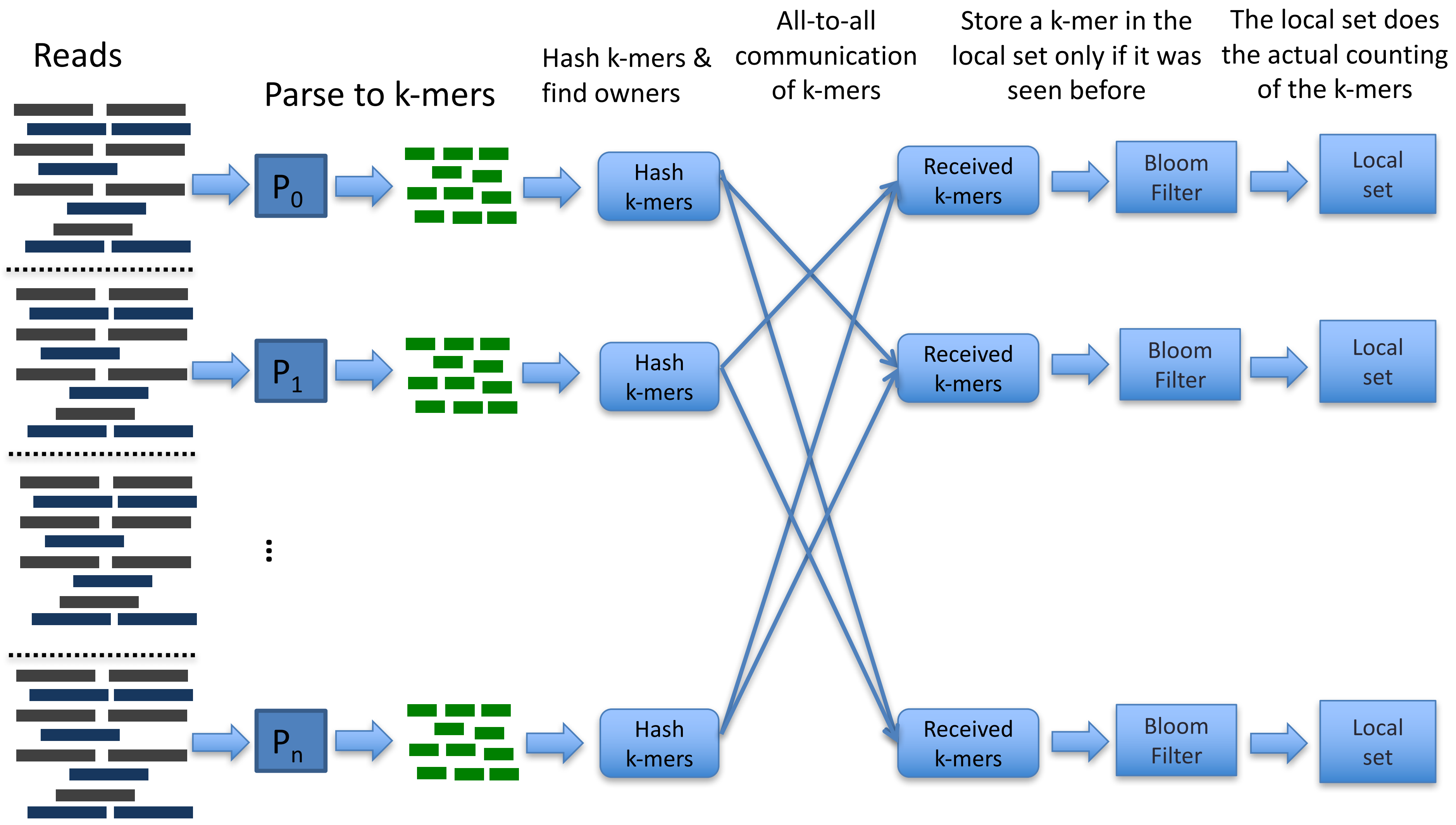
- UPC is a PGAS (**P**artitioned **G**lobal **A**ddress **S**pace) parallel language with **one-sided communication**.
- Core graph algorithms implemented in UPC (graph ⬄ hash table).
- We need the notion of **a huge global distributed hash table**.
- **Irregular access pattern** in the the distributed hash table
  - One-sided communication is handy!
- Portable implementation: can run any machine without change!
- **Result of this work***: Complete assembly of human genome in* ***8.4 minutes*** *using 15K cores*
- Original code required **2 days** and a large memory machine**.**

Parallel k-mer analysis: pass 1


Reads
Parse to k-mers
k-mer cardinality estimation
Parallel Reduction
Create local Bloom filters
P_0
P_1
P_n
Hyperloglog
Hyperloglog
Hyperloglog
G = Global cardinality estimation
Bloom Filter of size (G/n)
Bloom Filter of size (G/n)
Bloom Filter of size (G/n)

Also identify high-frequency k-mers at this step

Parallel k-mer analysis: pass 2


Reads
Parse to k-mers
Hash k-mers & find owners
All-to-all communication of k-mers
Store a k-mer in the local set only if it was seen before
The local set does the actual counting of the k-mers
P_0
P_1
P_n
Hash k-mers
Received k-mers
Bloom Filter
Local set

Why use a Bloom filter?

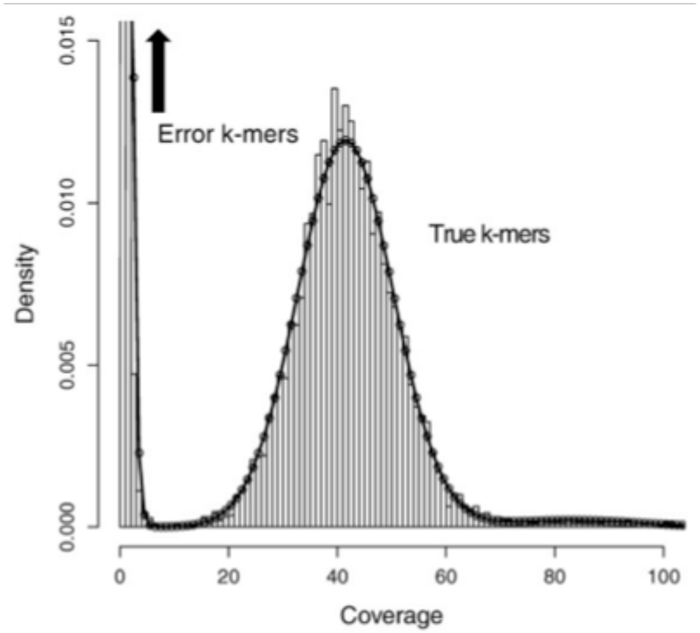
Bloom filter is a *probabilistic* data structure used for membership queries

- Given a bloom filter, we can ask:

"Have we seen this k-mer before?"

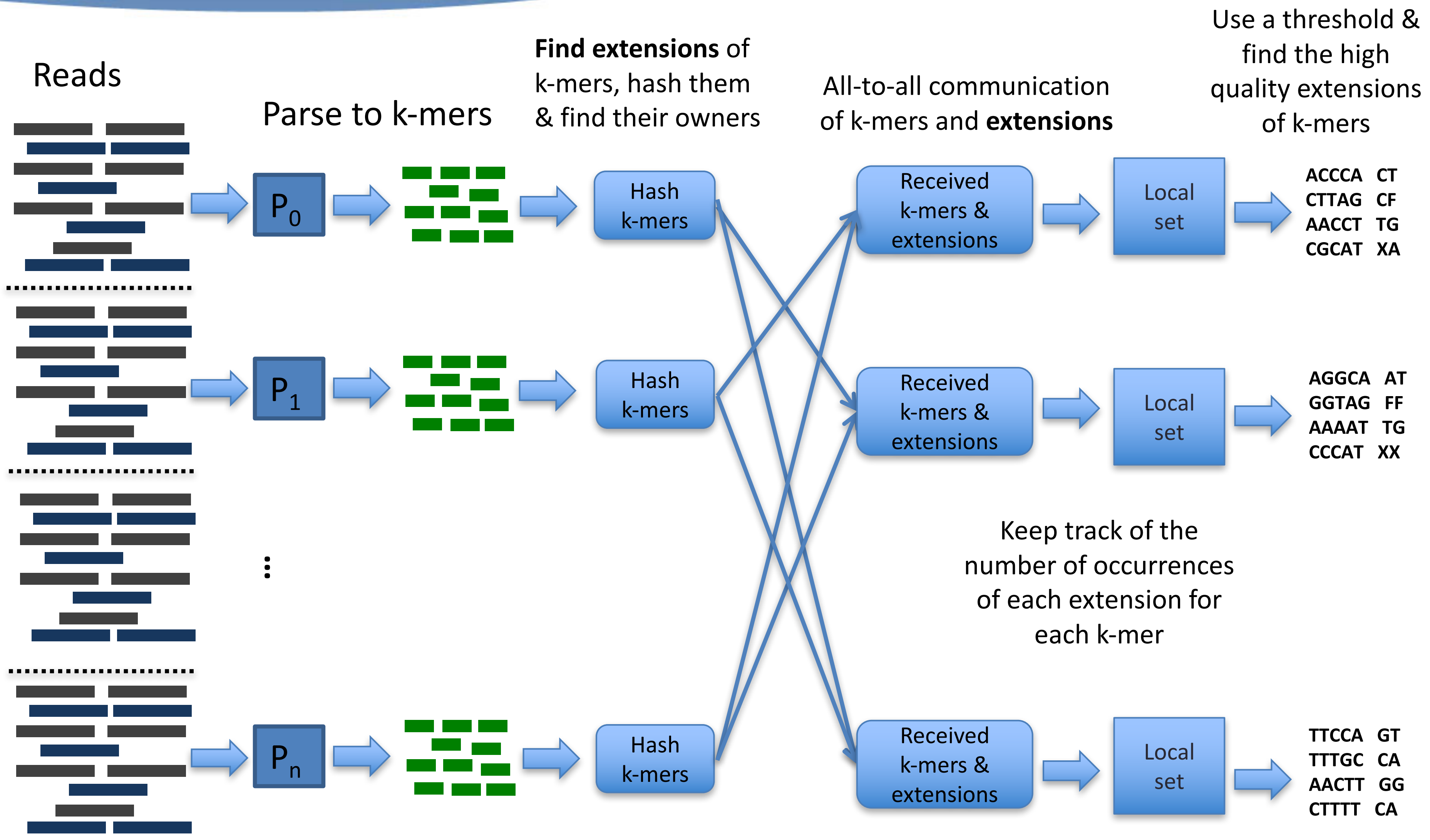
- **No false negatives.**
- May have false positives

(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

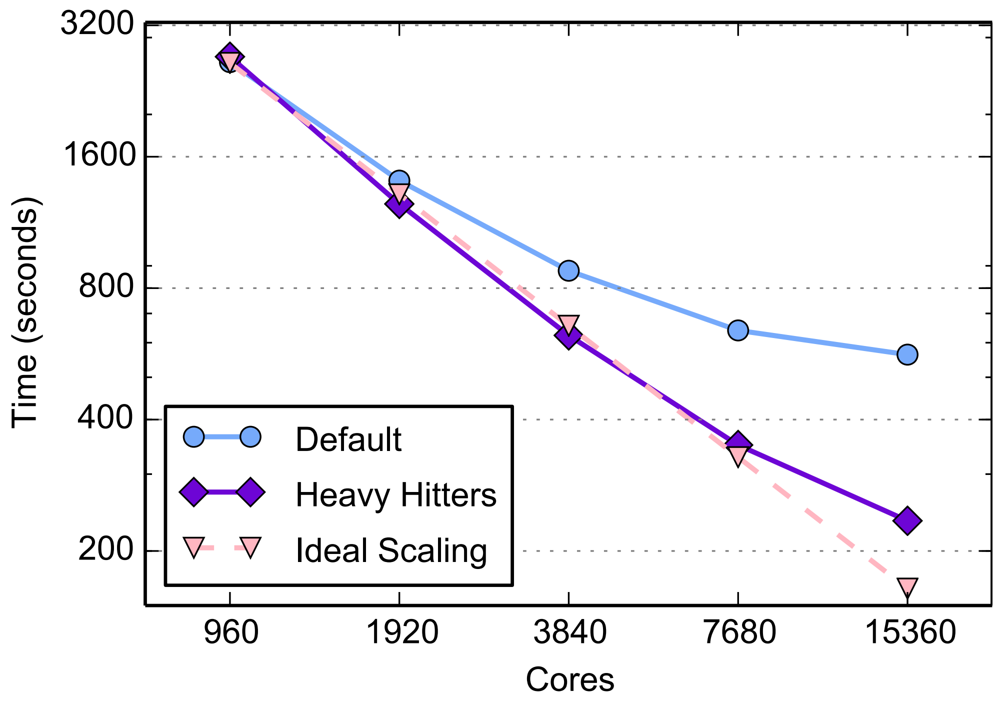

Reads
Parse to k-mers
Find extensions of k-mers, hash them & find their owners
All-to-all communication of k-mers and **extensions**
Use a threshold & find the high quality extensions of k-mers
P_0
P_1
P_n
Hash k-mers
Hash k-mers
Hash k-mers
Received k-mers & extensions
Received k-mers & extensions
Received k-mers & extensions
Local set
Local set
Local set
ACCCA CT
CTTAG CF
AACCT TG
CGCAT XA
AGGCA AT
GGTAG FF
AAAAT TG
CCCAT XX
TTCCA GT
TTTGC CA
AACTT GG
CTTTT CA
Keep track of the number of occurrences of each extension for each k-mer

High-frequency k-mers

Long-tailed distribution for genomes with repetitive content:

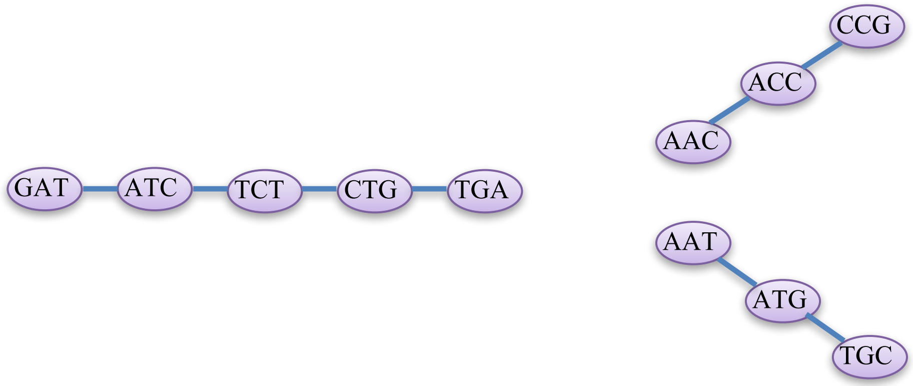
- The maximum count for any k-mer in the wheat dataset is **451 million**
- Our original scheme (SC'14) was "owner counts", after an all-to-all
- Counting an item w/ 451 million occurrences alone is **load imbalanced**

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

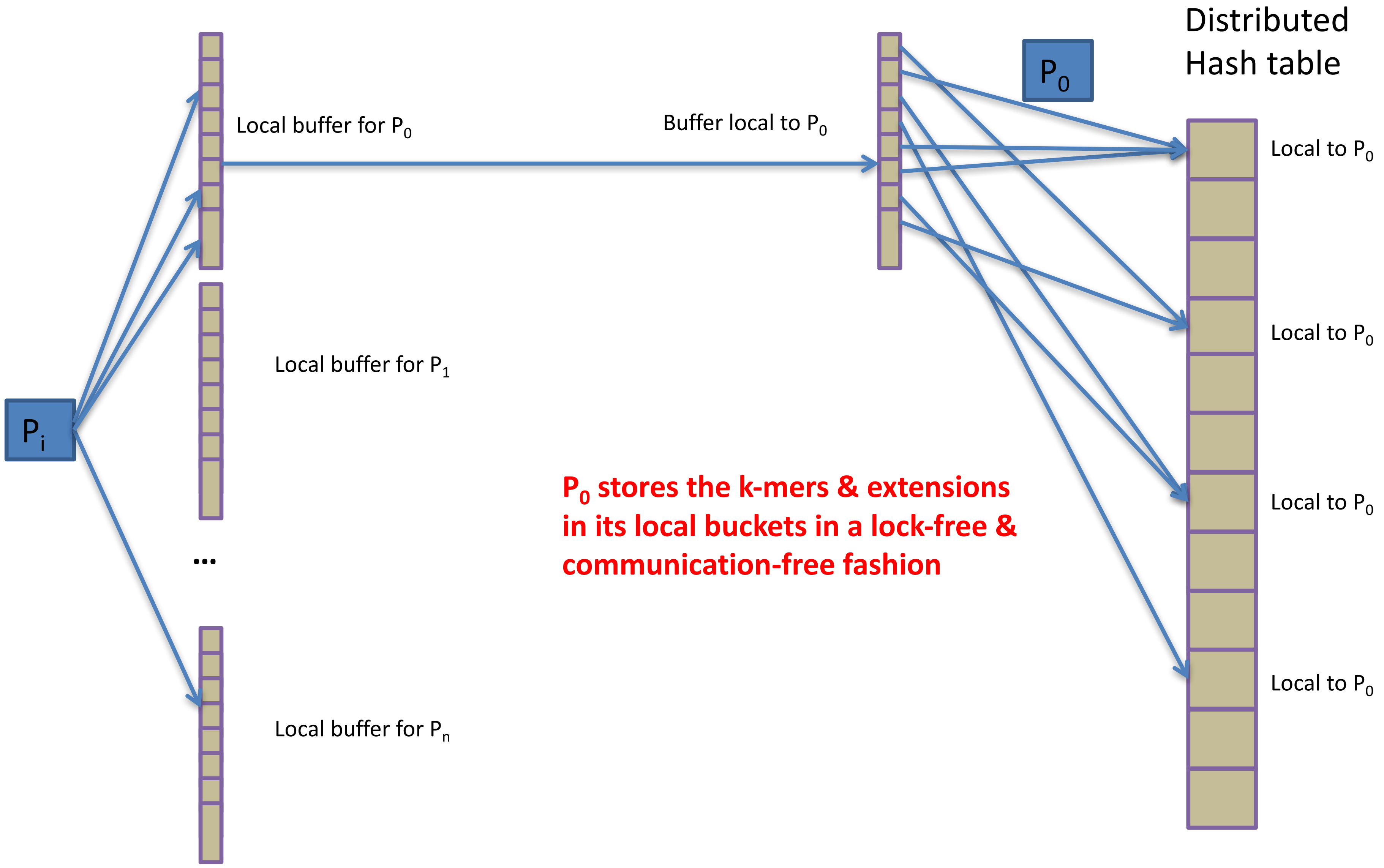


Parallel De Bruijn Graph Construction

- **Challenge 1:** The hash table that represents the de Bruijn graph is large (100s of GBs up to 10s of TBs !)
  - **Solution:** Distribute the graph over multiple processors. The global address space of UPC is handy!

- **Challenge 2:** Parallel hash tables construction introduces communication and synchronization costs
  - **Solution:** Aggregate messages to reduce number of messages and synchronization → **10x-20x performance improvement.**

Aggregating stores optimization

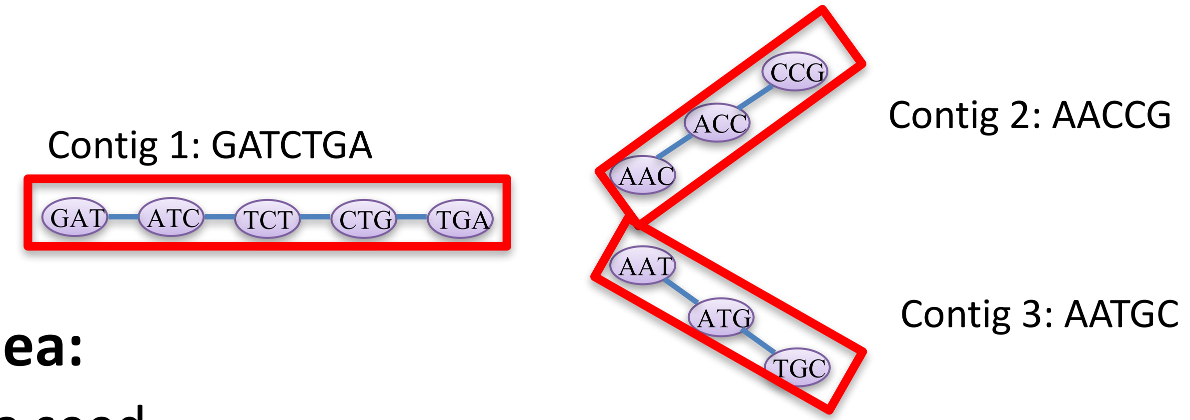


P_0 stores the k-mers & extensions in its local buckets in a lock-free & communication-free fashion

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:

CGTATTGCCAATGCAACGTATCATGGCCAATCCGAT

P_i knows that forward extension is A

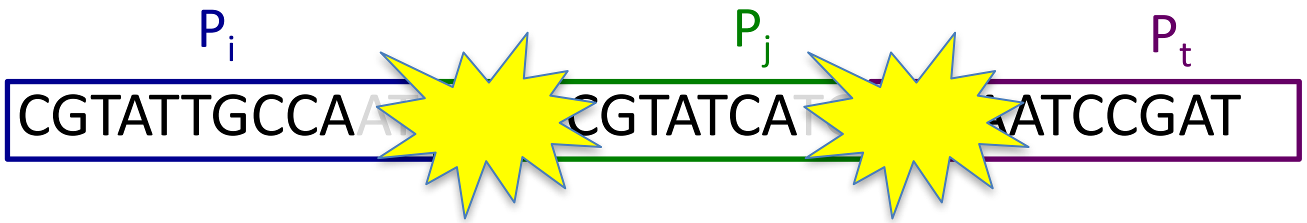
P_i uses the last k-1 bases and the forward extension and forms: CAACGTATCA

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



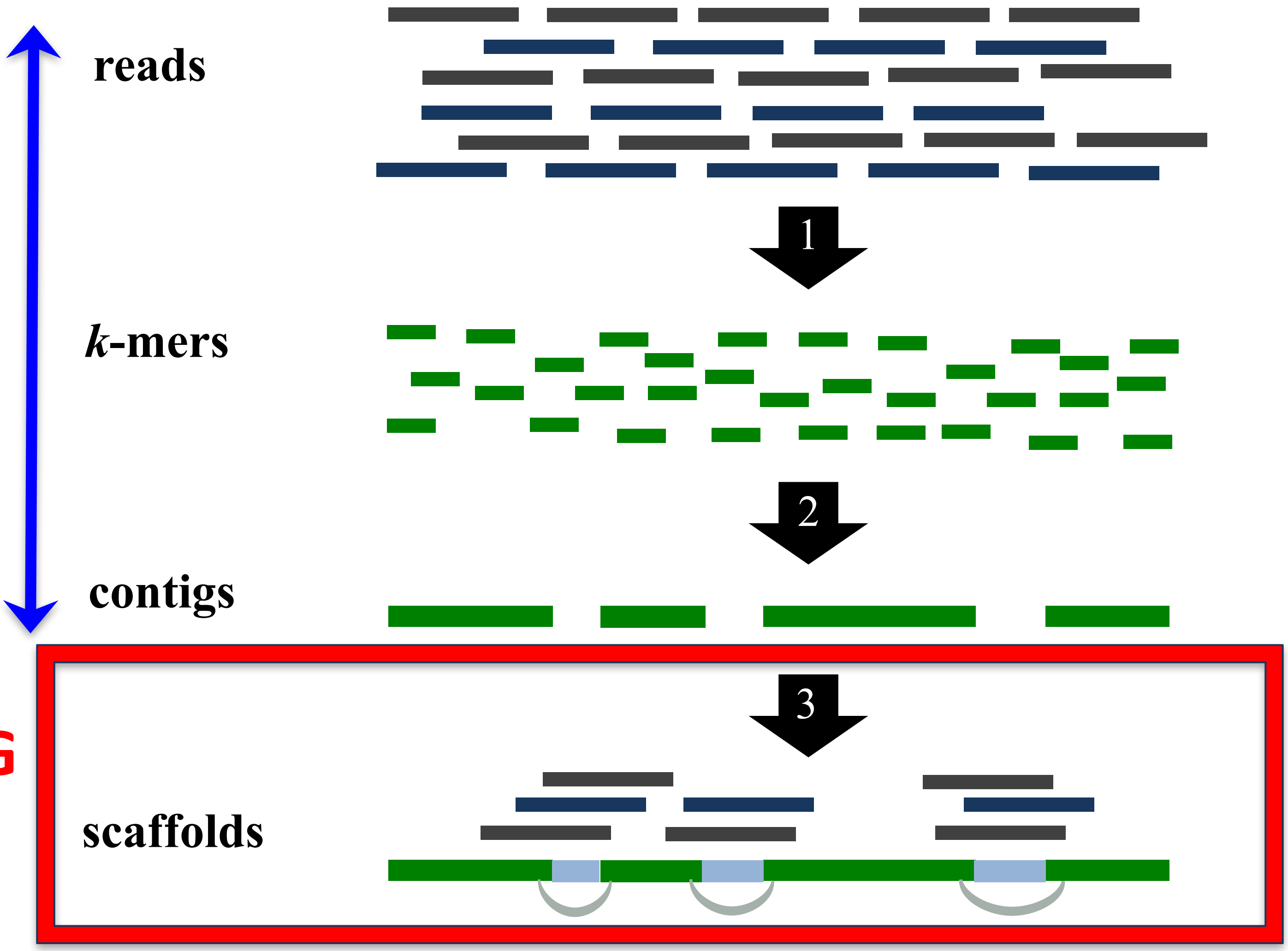
However, processors P_i, P_j and P_t might have picked initial seeds from the same UU contig

- 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

Alignment for De novo Genome Assembly

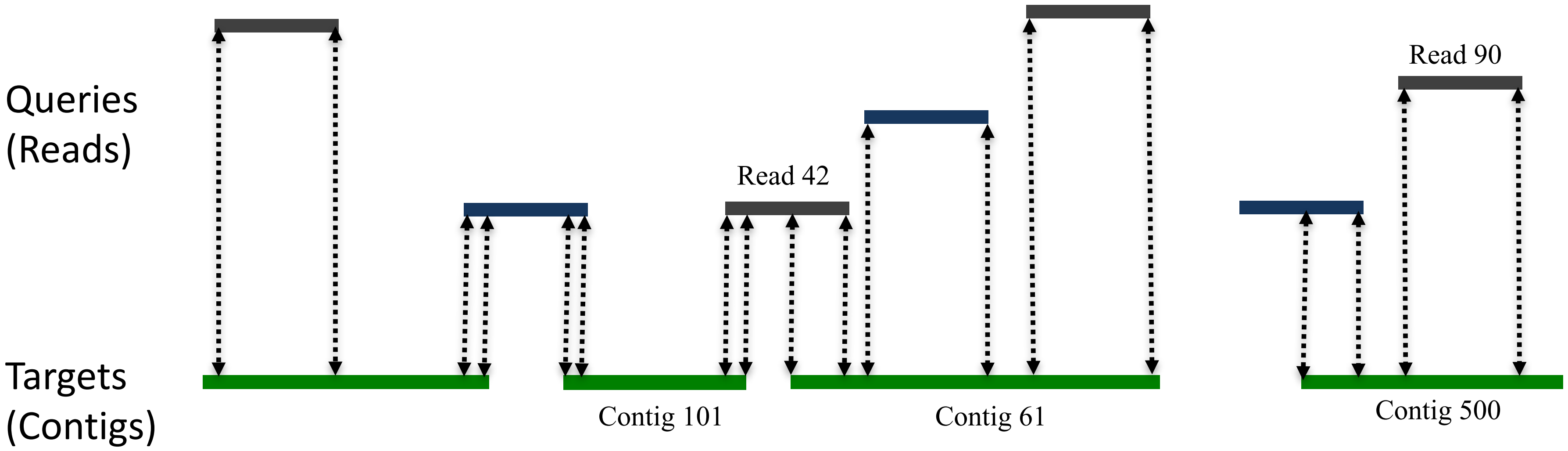
To be reused for metagenomics w/ modifications

PARALLELIZING SCAFFOLDING



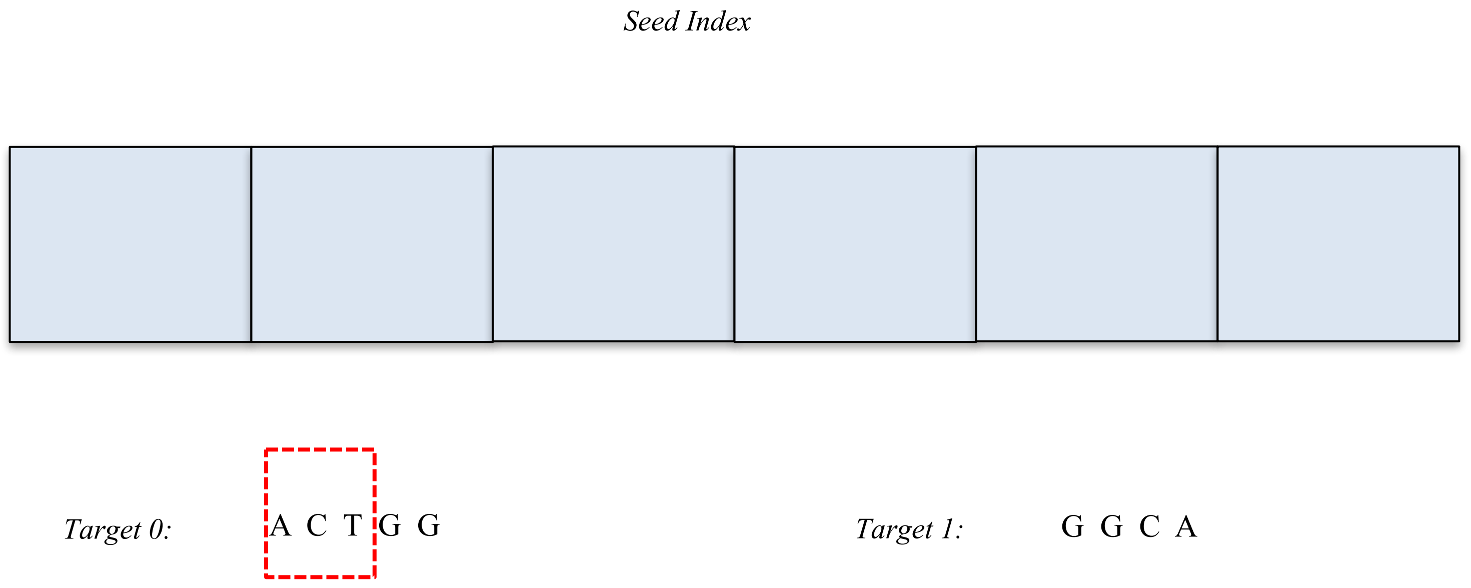
Aligning queries to target sequences

- 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**

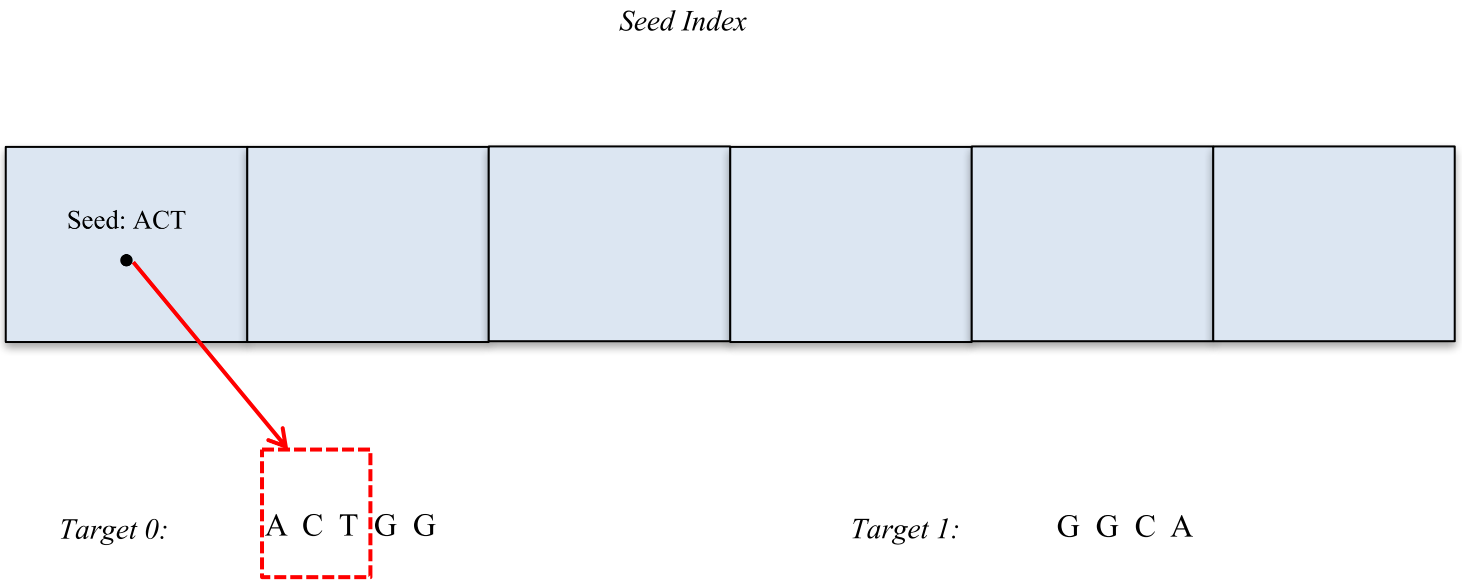


Read ID	start-pos	end-pos	Contig ID	start-pos	end-pos
Read 42	1	4	Contig 101	152	155
Read 42	130	150	Contig 61	1	21
Read 90	1	150	Contig 500	101	250

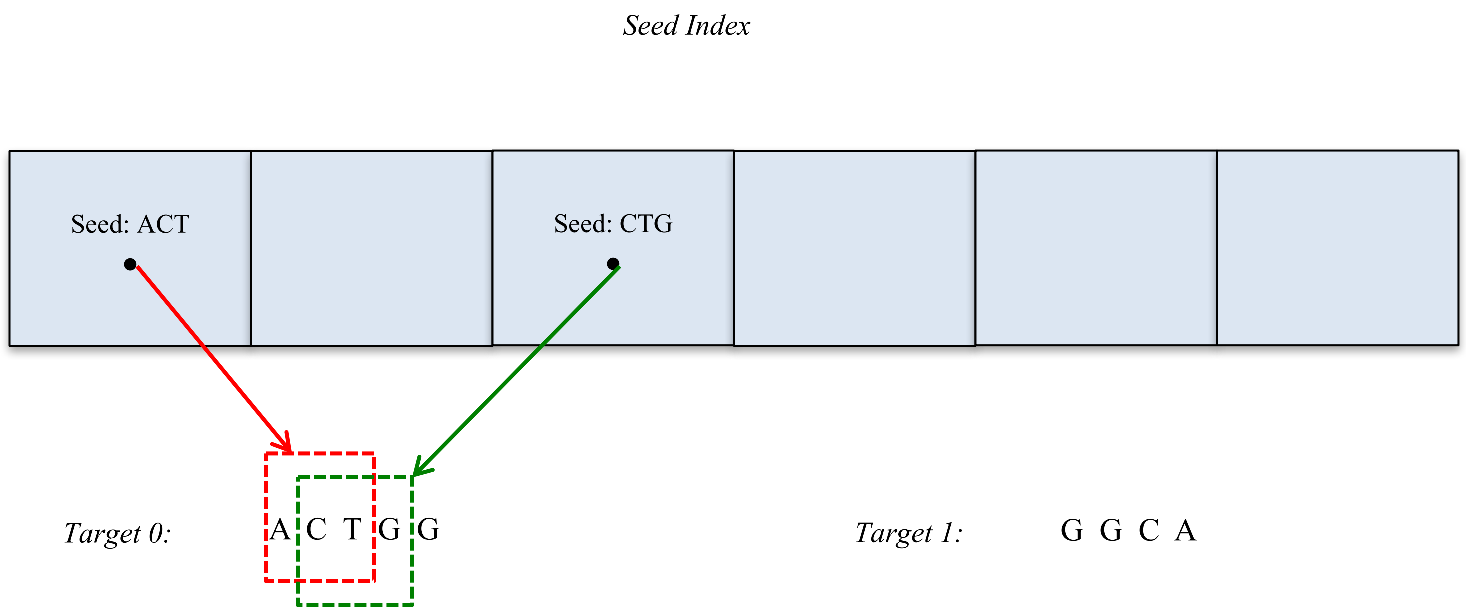
Building seed index



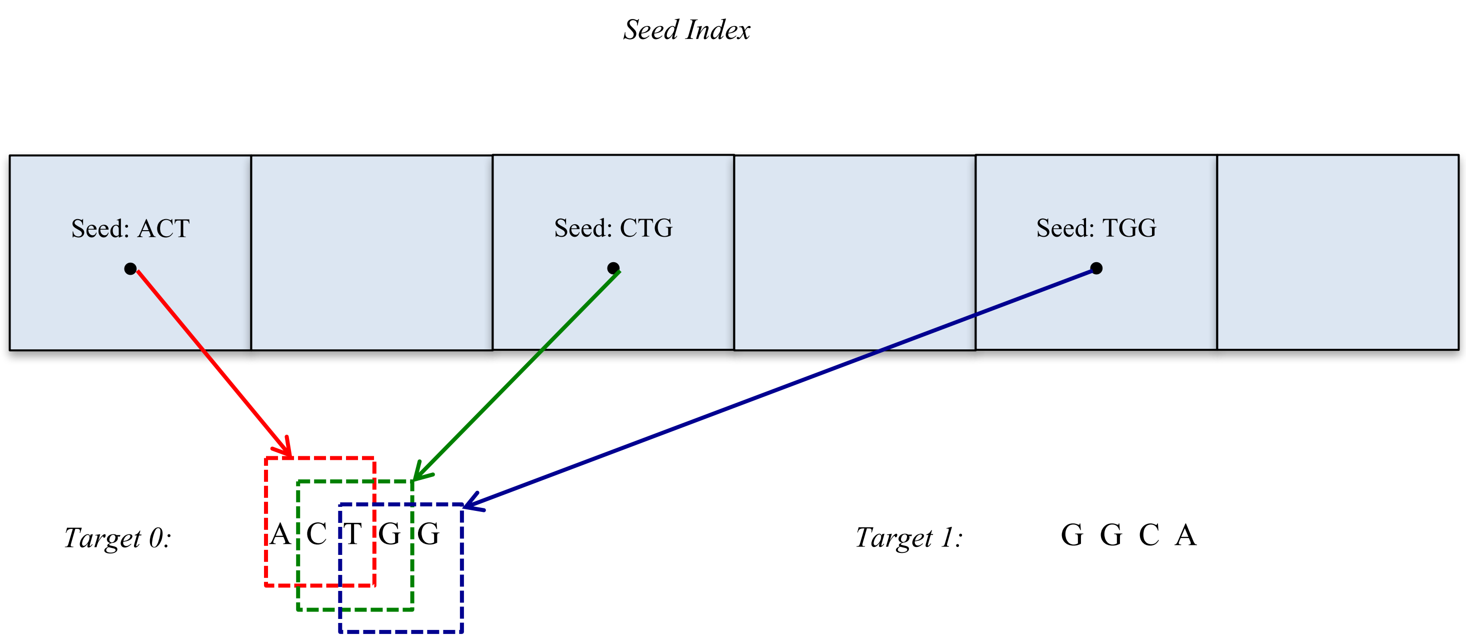
Building seed index



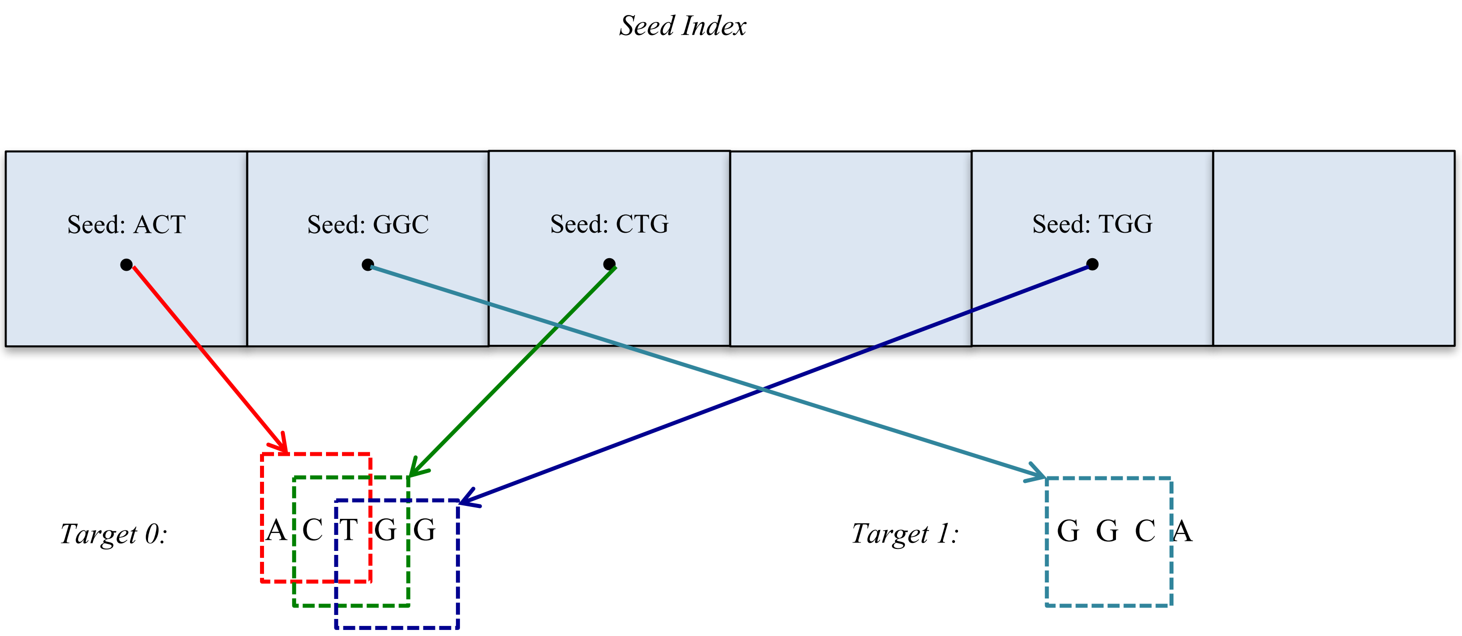
Building seed index



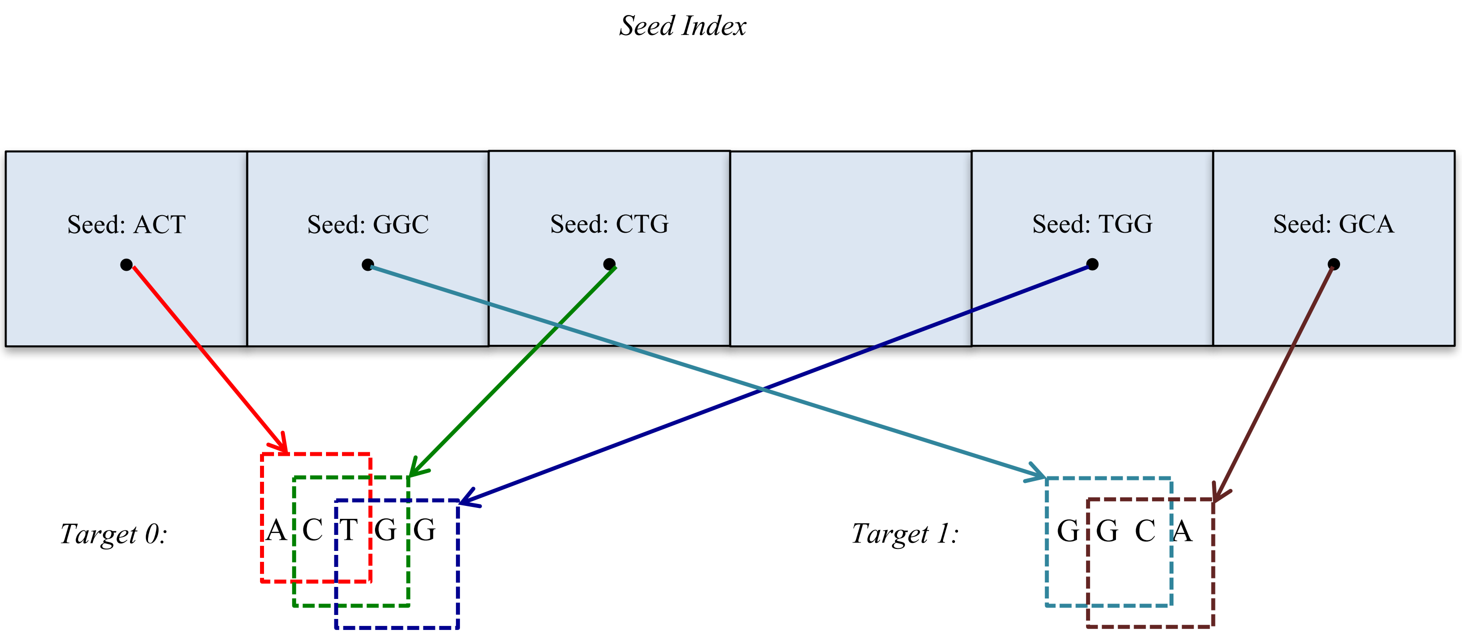
Building seed index



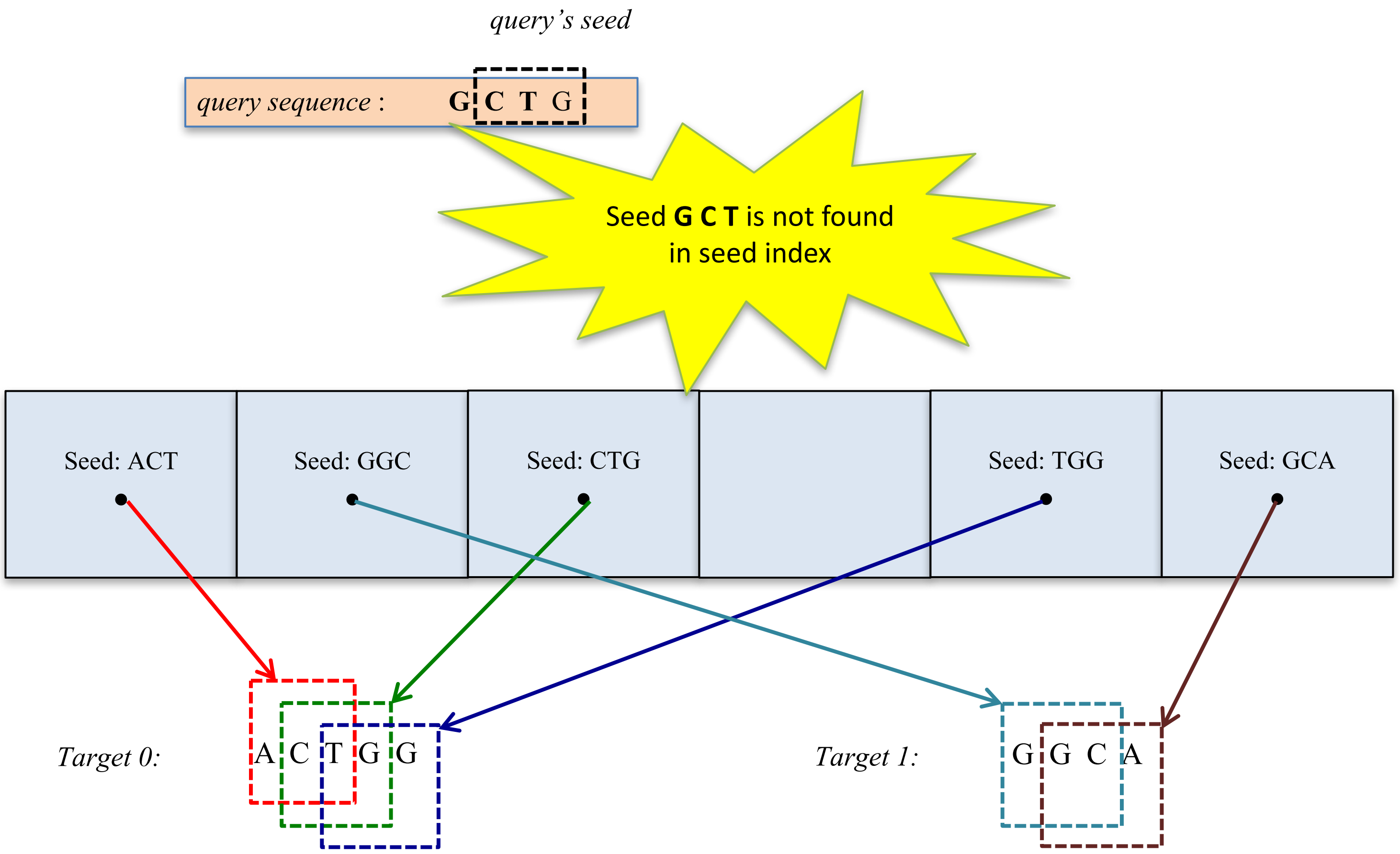
Building seed index



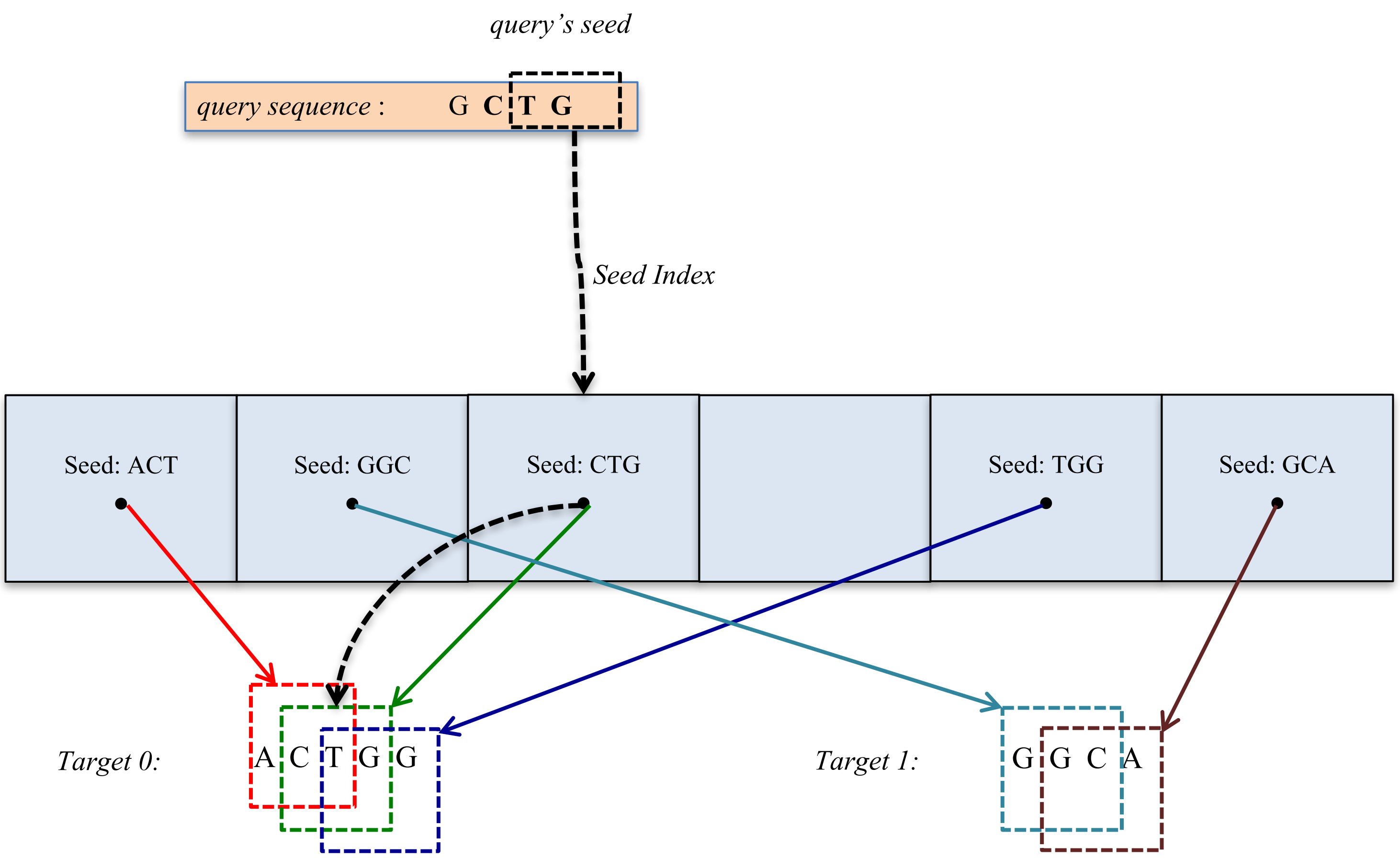
Building seed index



Lookup seed index

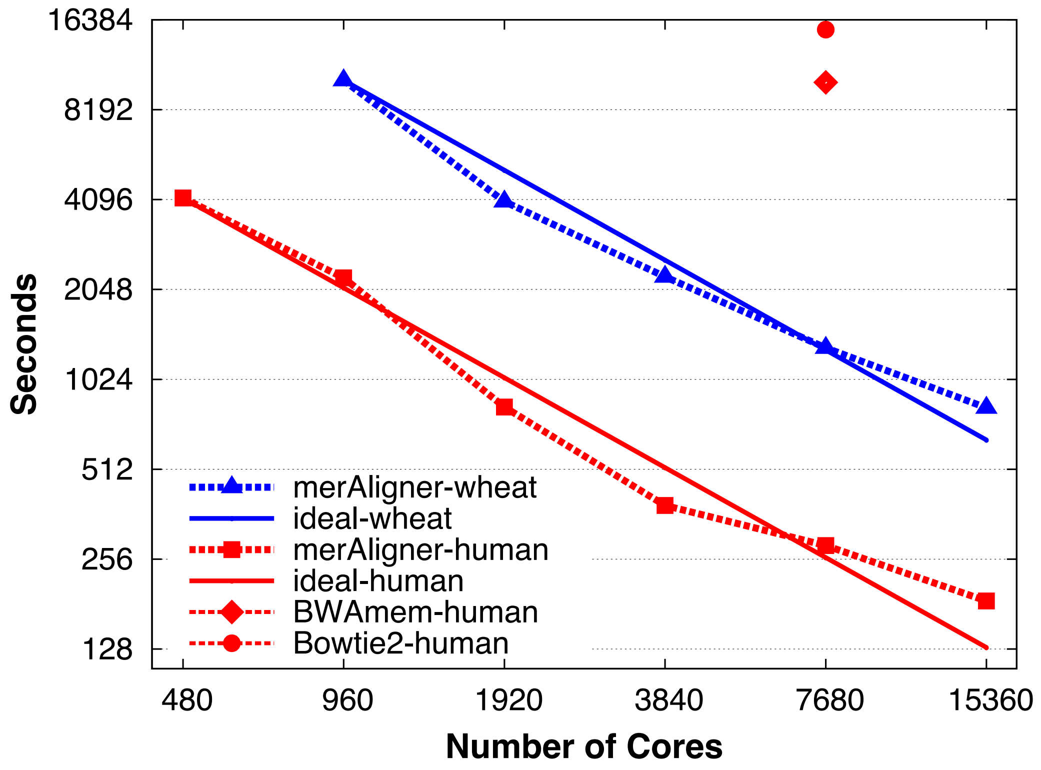


Lookup seed index



Parallel Genome Alignment for De novo Assembly

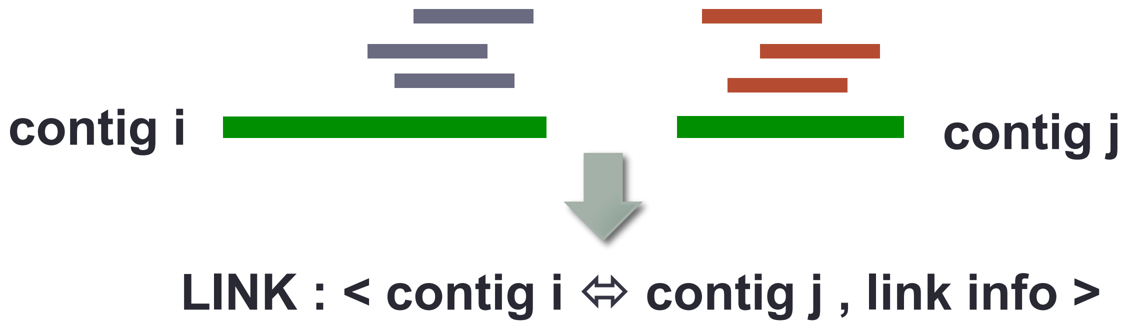
- In de novo assembly, billions of reads must be aligned to contigs
- First aligner to parallelize the seed index construction ("fully" parallel)



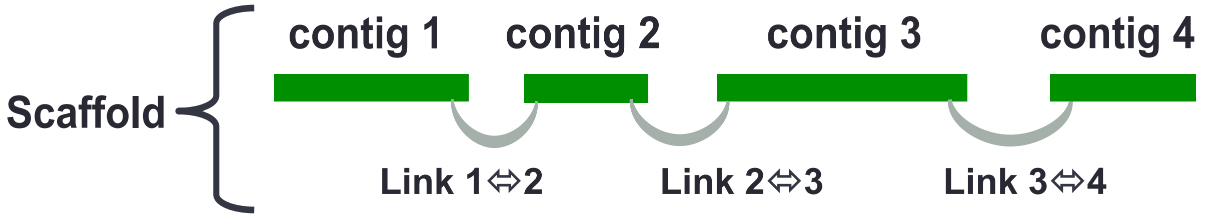
Evangelos Georganas, Aydın Buluç, Jarrod Chapman, Leonid Oliker, Daniel Rokhsar, and Katherine Yelick. **meraligner: A fully parallel sequence aligner**. In Proceedings of the IPDPS, 2015.

Scaffolding beyond alignment

- **High-level idea:** Leverage read-to-contig information to generate *links* among contigs.
  - Distributed hash tables to index the link information.

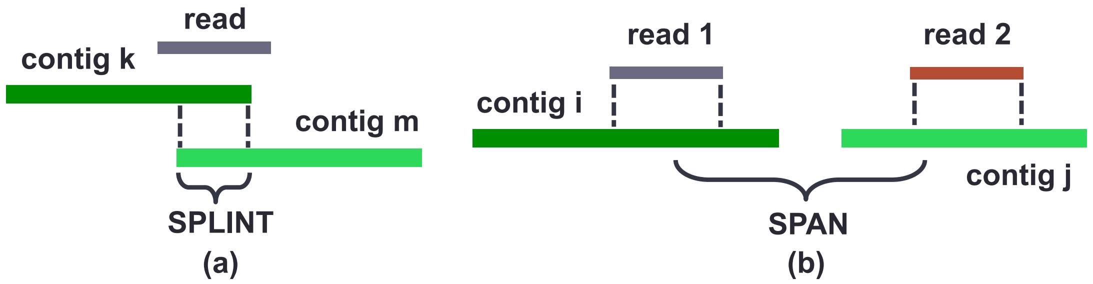


- Form a contig graph by using the links and traverse it to form scaffolds.



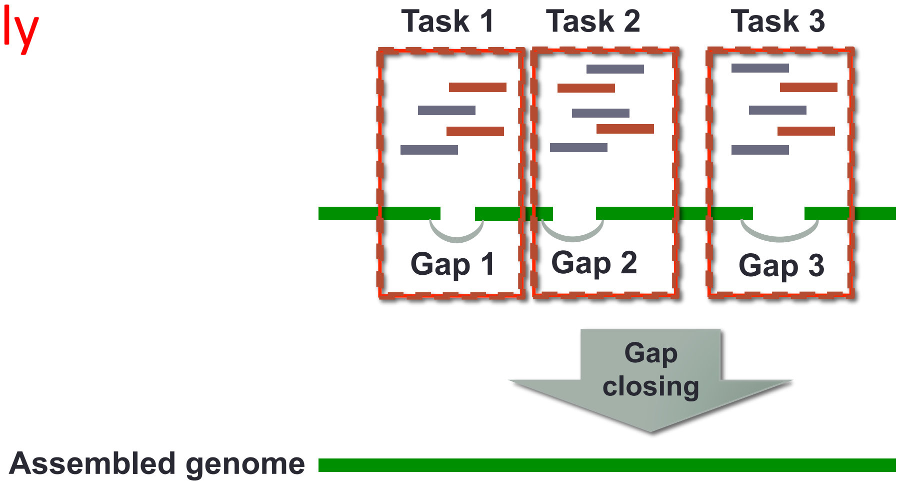
Scaffolding beyond alignment

- Computing contig depths and termination states
- Contig bubble identification (*for diploid genomes*)
- Ordering and orientation of contigs (*inherently serial as implemented*)
- Insert size estimation
- Locating ***splints*** and ***spans***
- Contig link generation
- Gap closing



Conceptually: Mini/local assembly

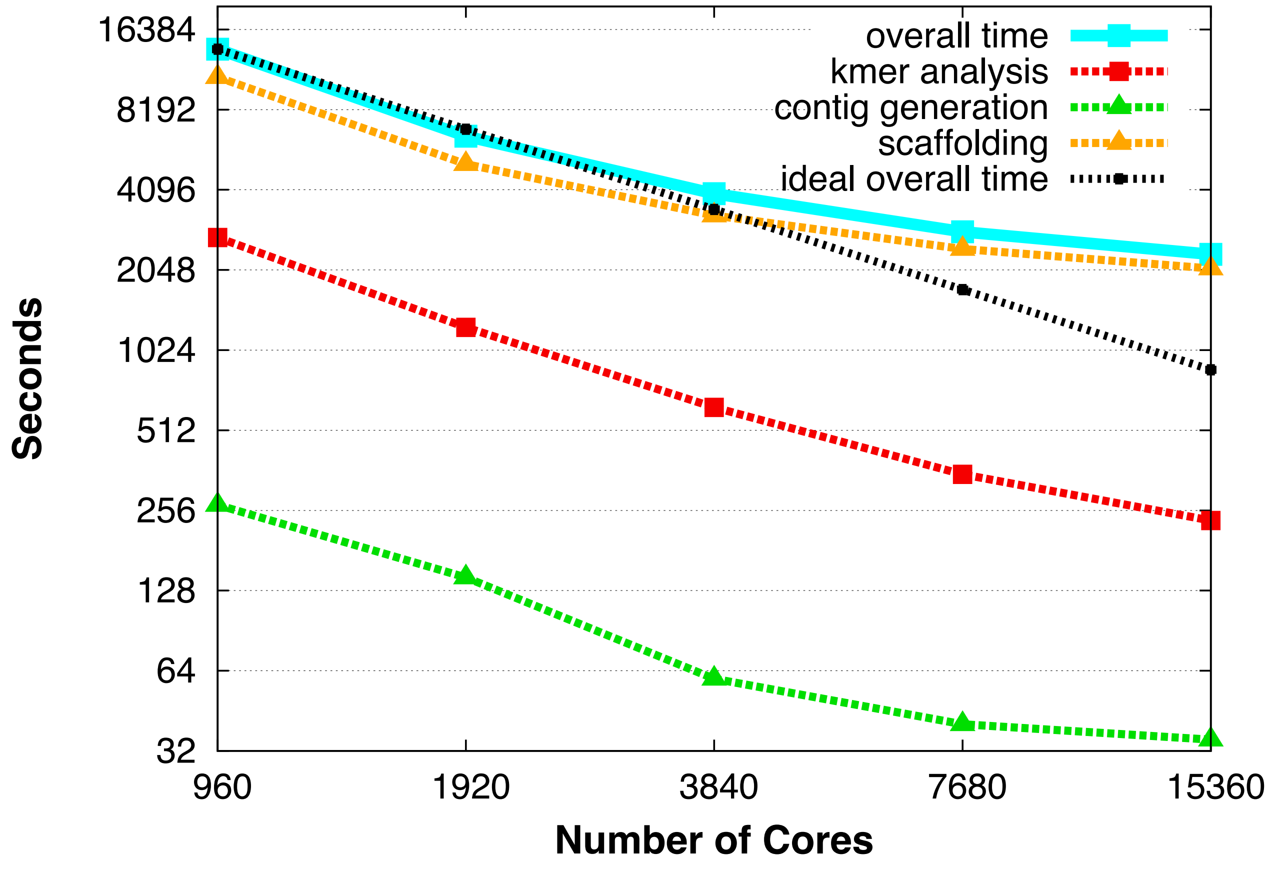
– embarrassingly parallelizable -



Tools employed:

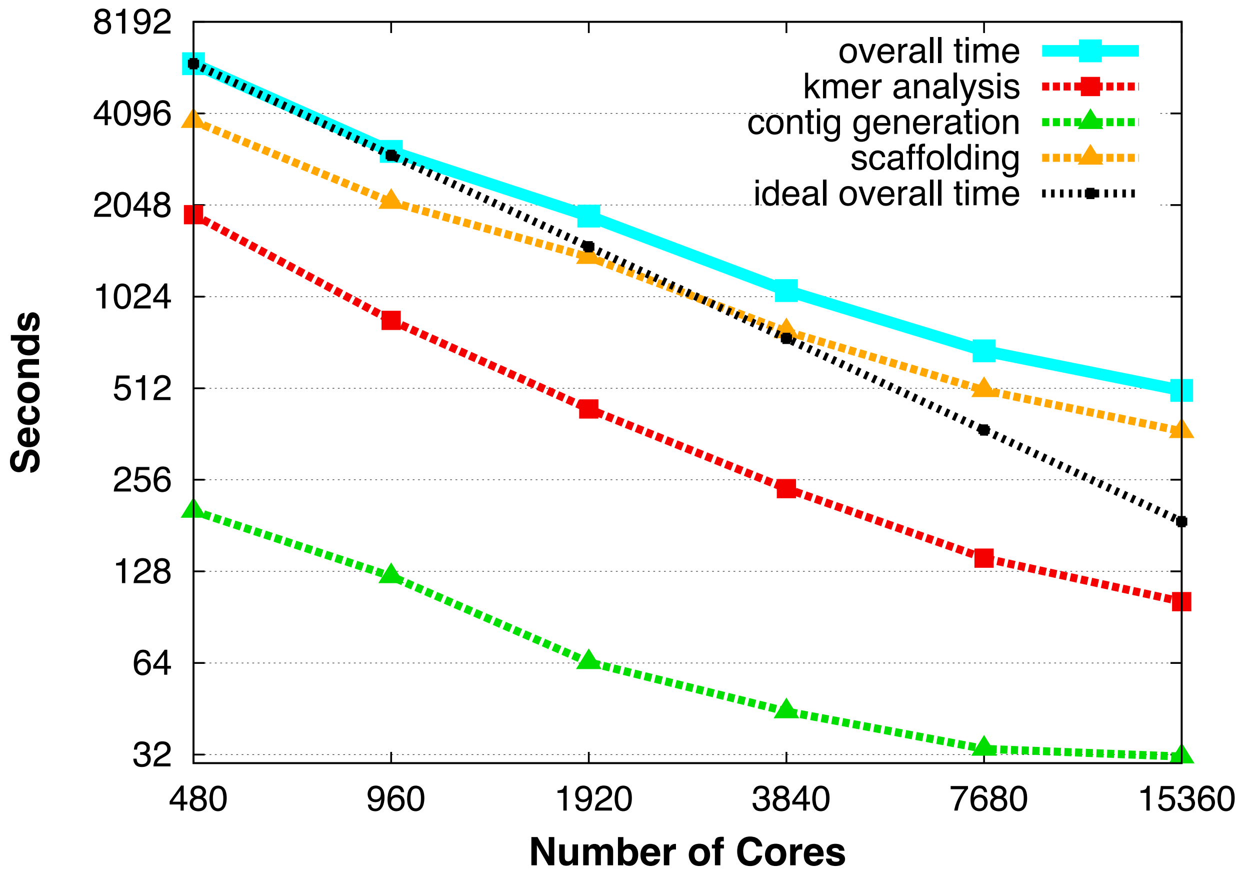
- **Distributed hash tables**
- **Speculative execution**

Strong scaling (wheat genome) on Cray XC30



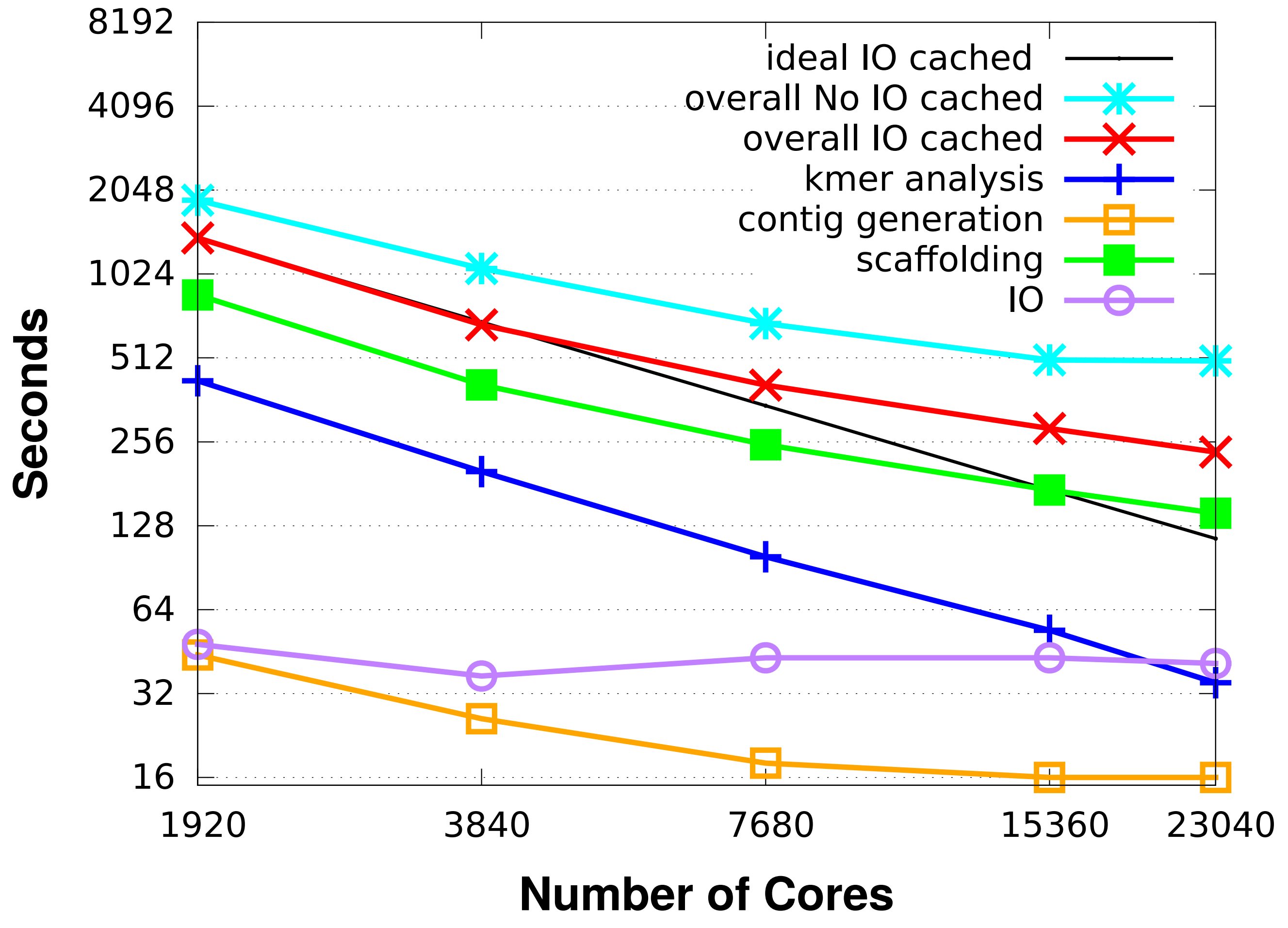
- Complete assembly of wheat genome in **39 minutes (15K cores).**
- Original Meraculous would require (projected time) **a week (~300x slower)** and a shared memory machine with 1TB memory.

Strong scaling (human genome) on Cray XC30 @SC'15



- Complete assembly of human genome in **8.4 minutes (15K cores).**
- **350x speedup over** original Meraculous (took **2,880 minutes** and a large shared memory machine).

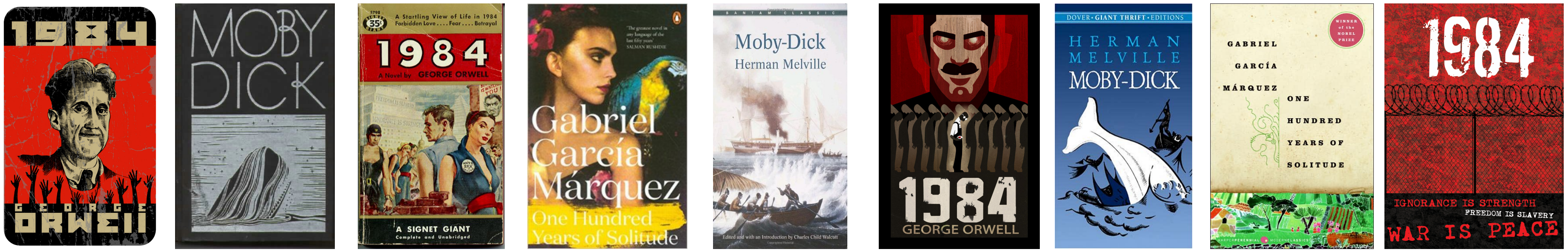
Strong scaling (human genome) on Cray XC30 @Now



- Complete assembly of human genome in **4 minutes using 23K cores.**
- **700x speedup over** original Meraculous (took **2,880 minutes** and a large shared memory machine).

How about metagenomes?

- Instead of multiple copies of the same book, how you have the whole library to assemble!

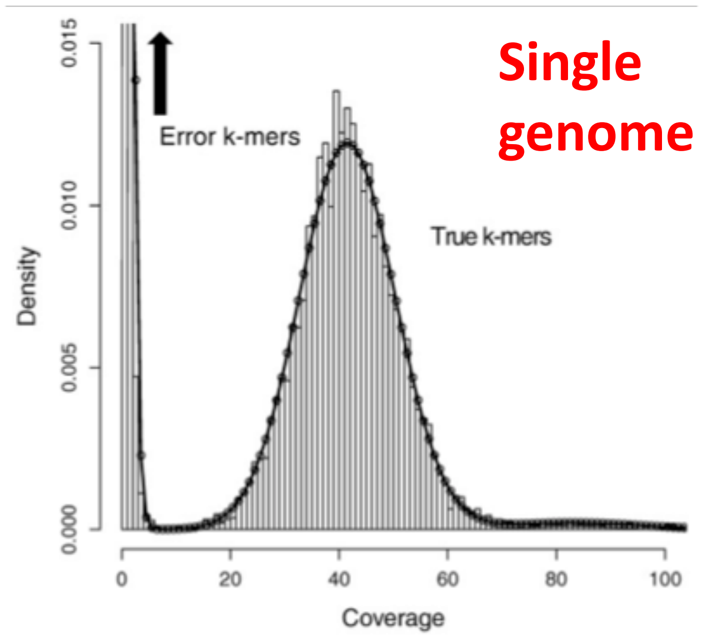


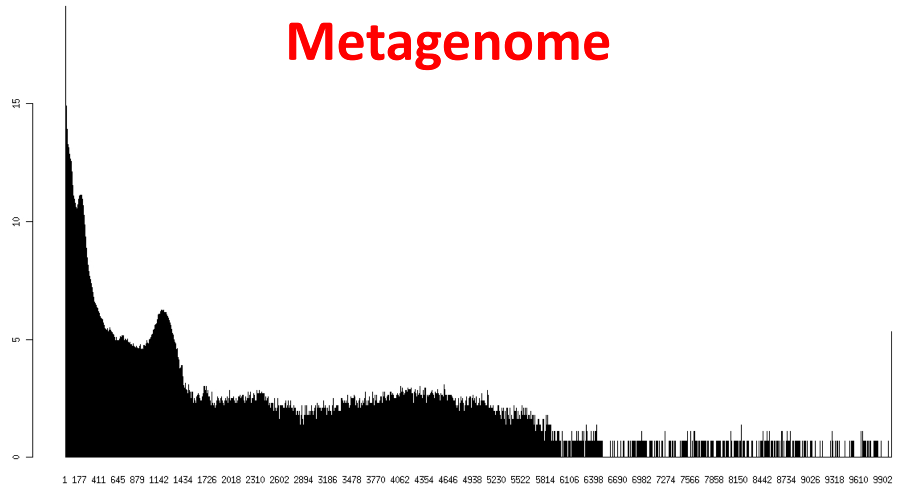
- 1984 is the "dominant species" in this sample.
- Too easy: there are also previously unknown books in the mix.

Option 1: ***Bin the reads*** to genomes; run single genome assembly
☹ Reads are too short to contain enough information for binning.
Option 2: Generate contigs and ***bin the contigs***
☹ How do you eliminate errors for low-coverage organisms?

Metagenome challenges

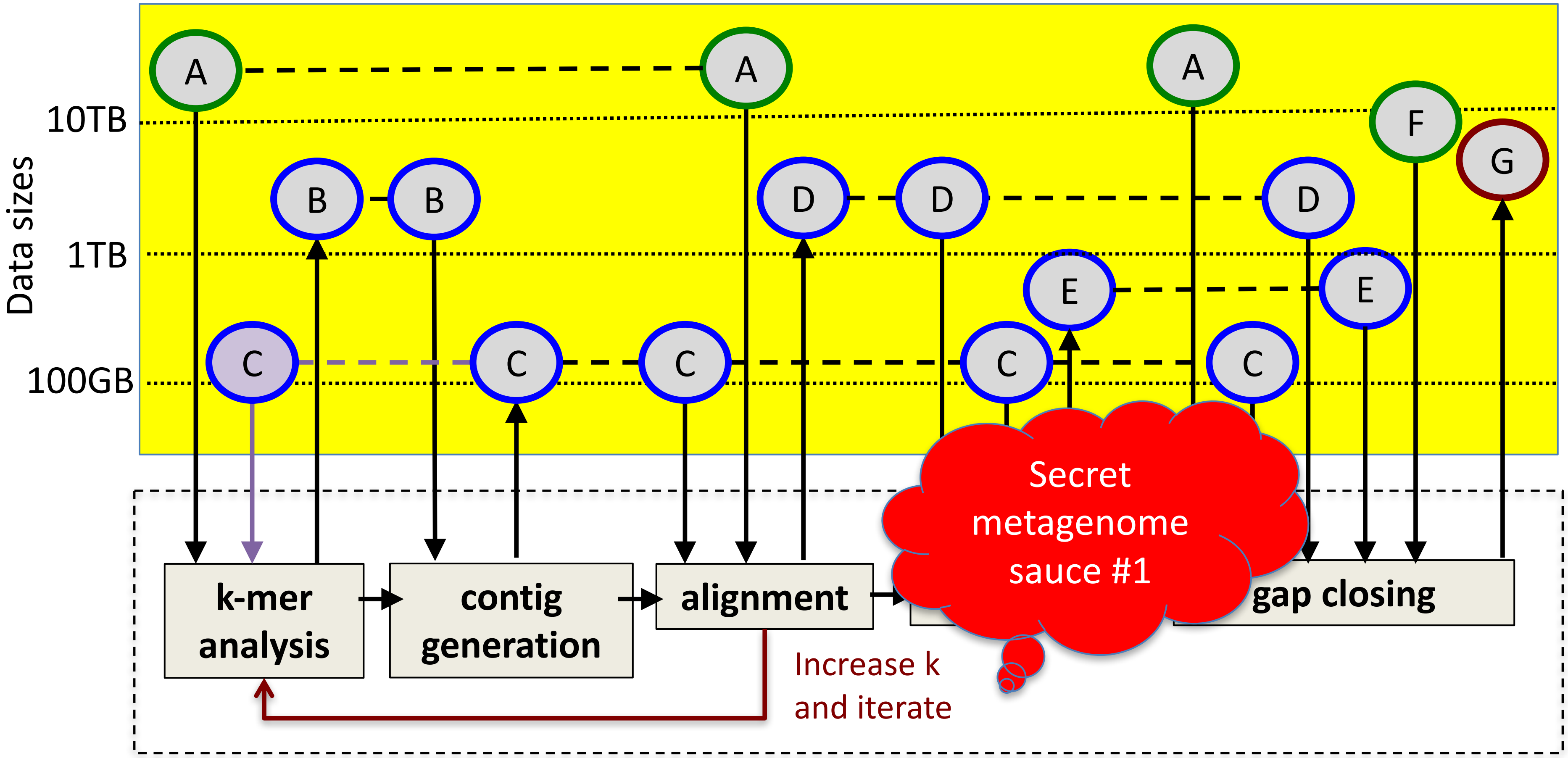
- Why does metagenome assembly benefits from explicit error correction when single genome assembly does not?
+ **Uneven sequencing depth:** Errors in high-depth regions might be more frequent than true k-mers in low-depth regions.
- Who cares about low-depth regions?
+ In fact, that's what matters most.



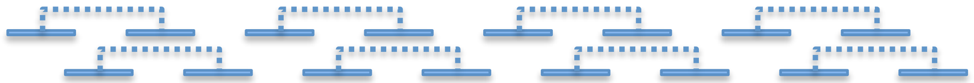


Adapting HipMer to metagenomes

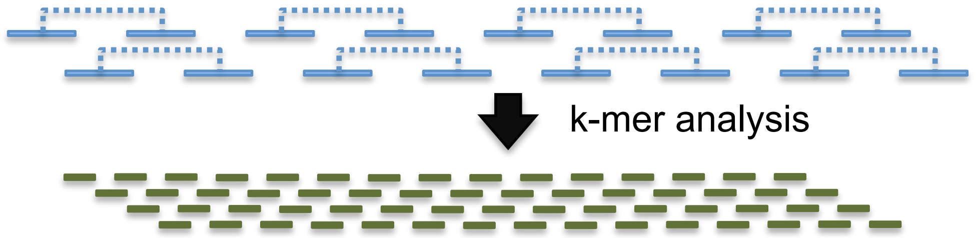
Inputs: (A) Short reads and (F) long insert (mate paired) libraries
Intermediate: (B) Error-free k-mers w/ extensions (a.k.a. UFX), (C) contigs…
Output: (G) final genome scaffolds



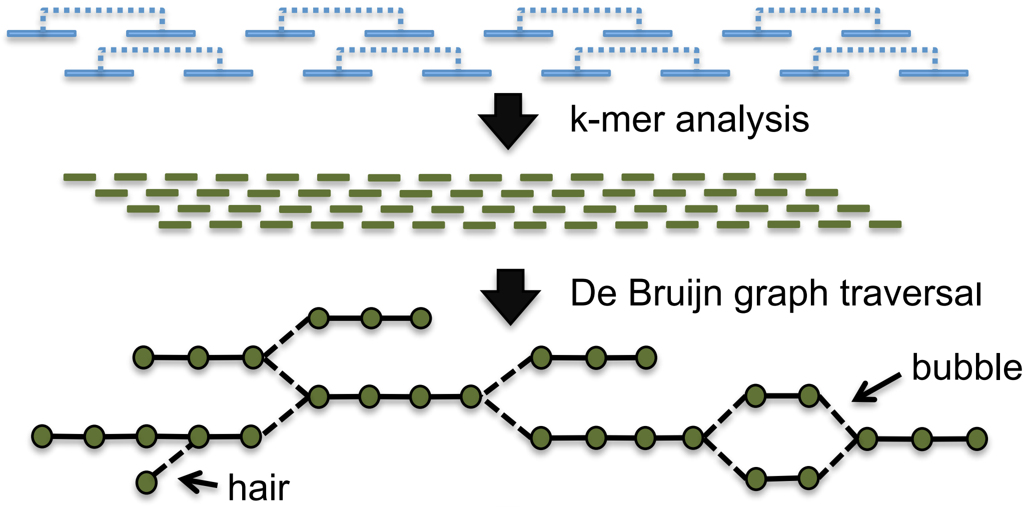
Iterative contig generation



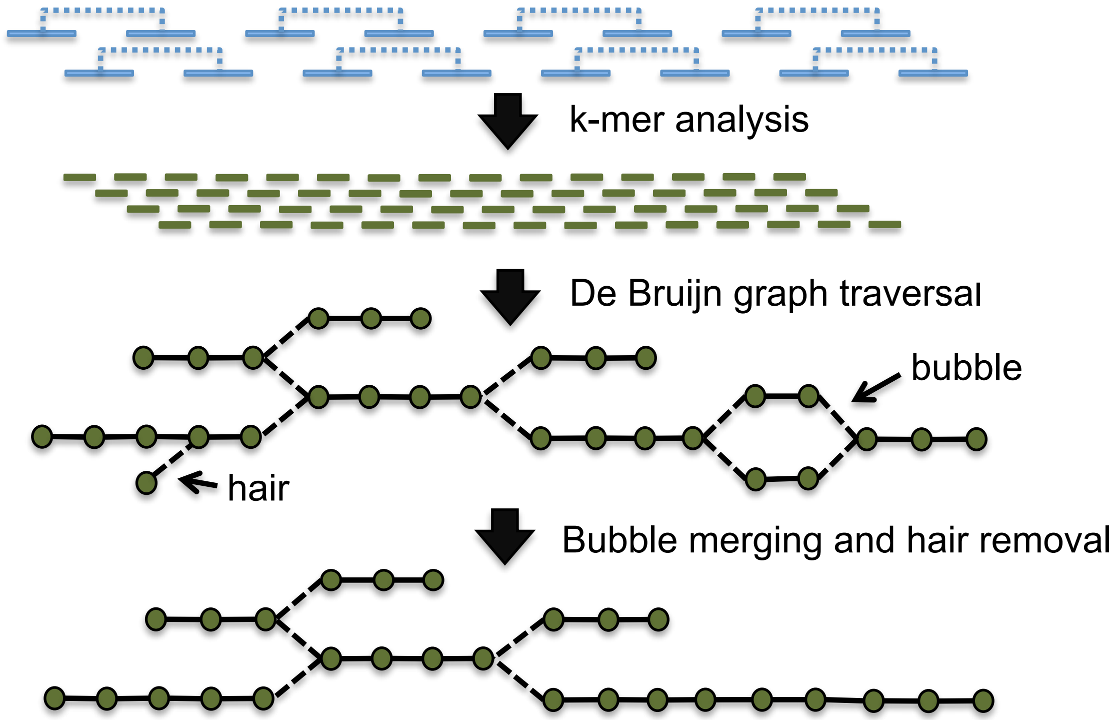
Iterative contig generation



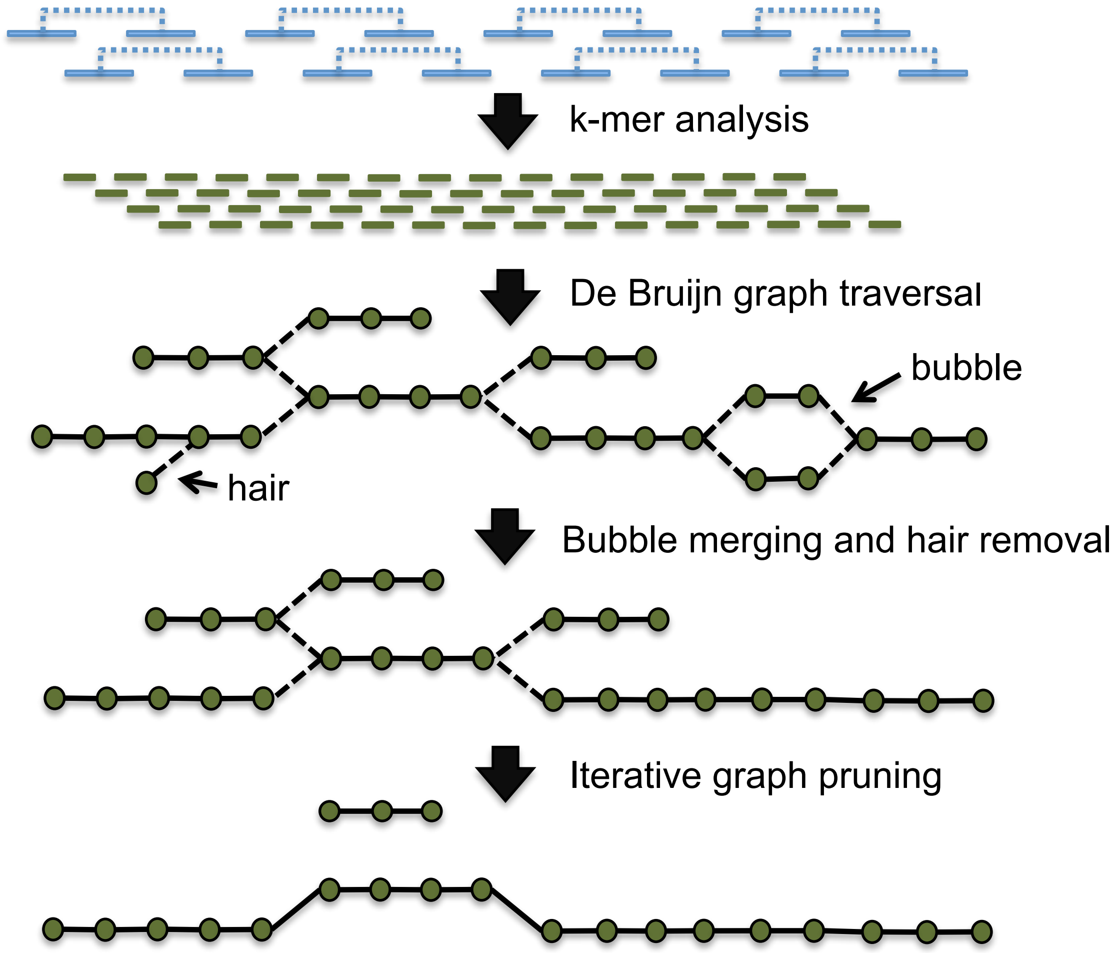
Iterative contig generation



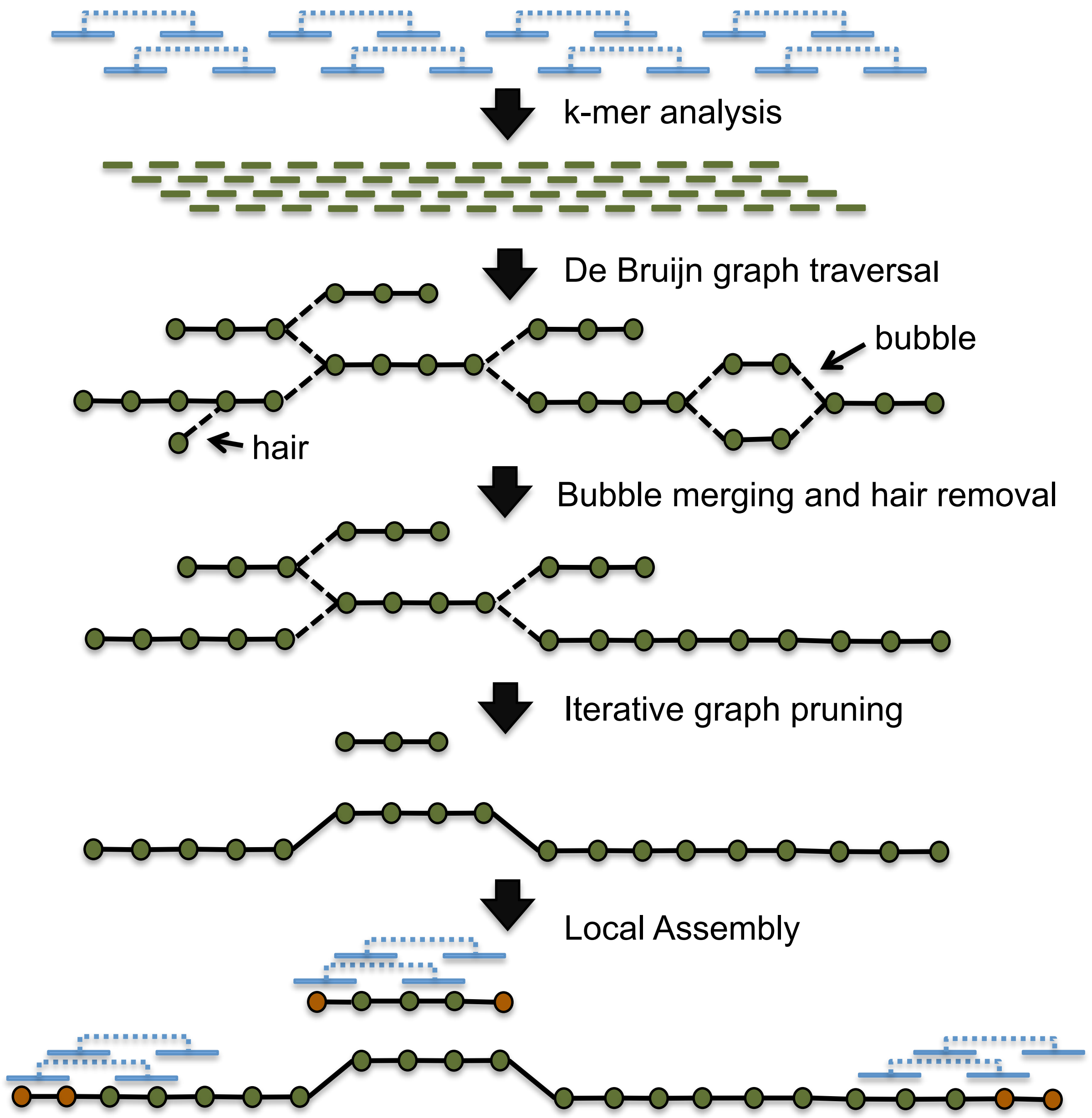
Iterative contig generation



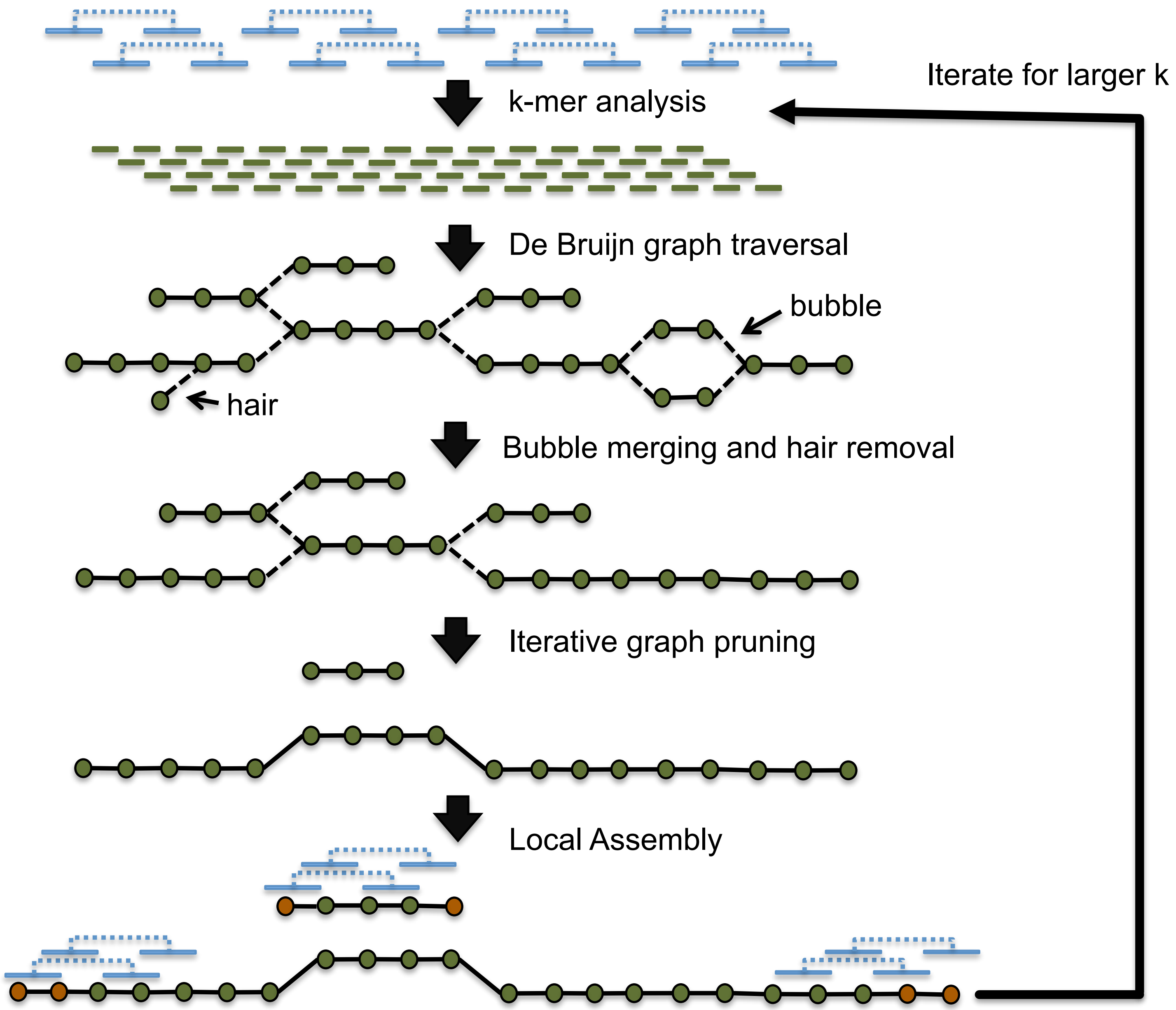
Iterative contig generation



Iterative contig generation



Iterative contig generation



HipMer on a MOCK community

- Mix of 25 bacteria with different abundancies (JGI dataset)
- Dataset size ~ 100 GBytes

Statistics	metaHipMer	metaSPAdes
# contigs	**2,670**	5,100
Total length > 10 kbp	**92,245,198**	92,152,728
Total length > 50 kbp	69,493,125	**77,292,121**
Misassemblies	**58**	134
Mismatches per 100 kbp	**3.48**	77.05
Genomes fraction (%)	**92.10**	91.10

Summary

- HipMer’s core algorithms scale to tens of thousands of cores and yield performance improvements from days/weeks down to minutes.
- HipMer breaks the hardware limitations by enabling distributed-memory scaling
- Use of de novo assembly in time sensitive applications like precision medicine is no more formidable!
- Source release of HipMer : https://sourceforge.net/projects/hipmer/
- Ongoing work: Adapt HipMer for metagenomic analysis (some hints and preliminary results given in this talk).