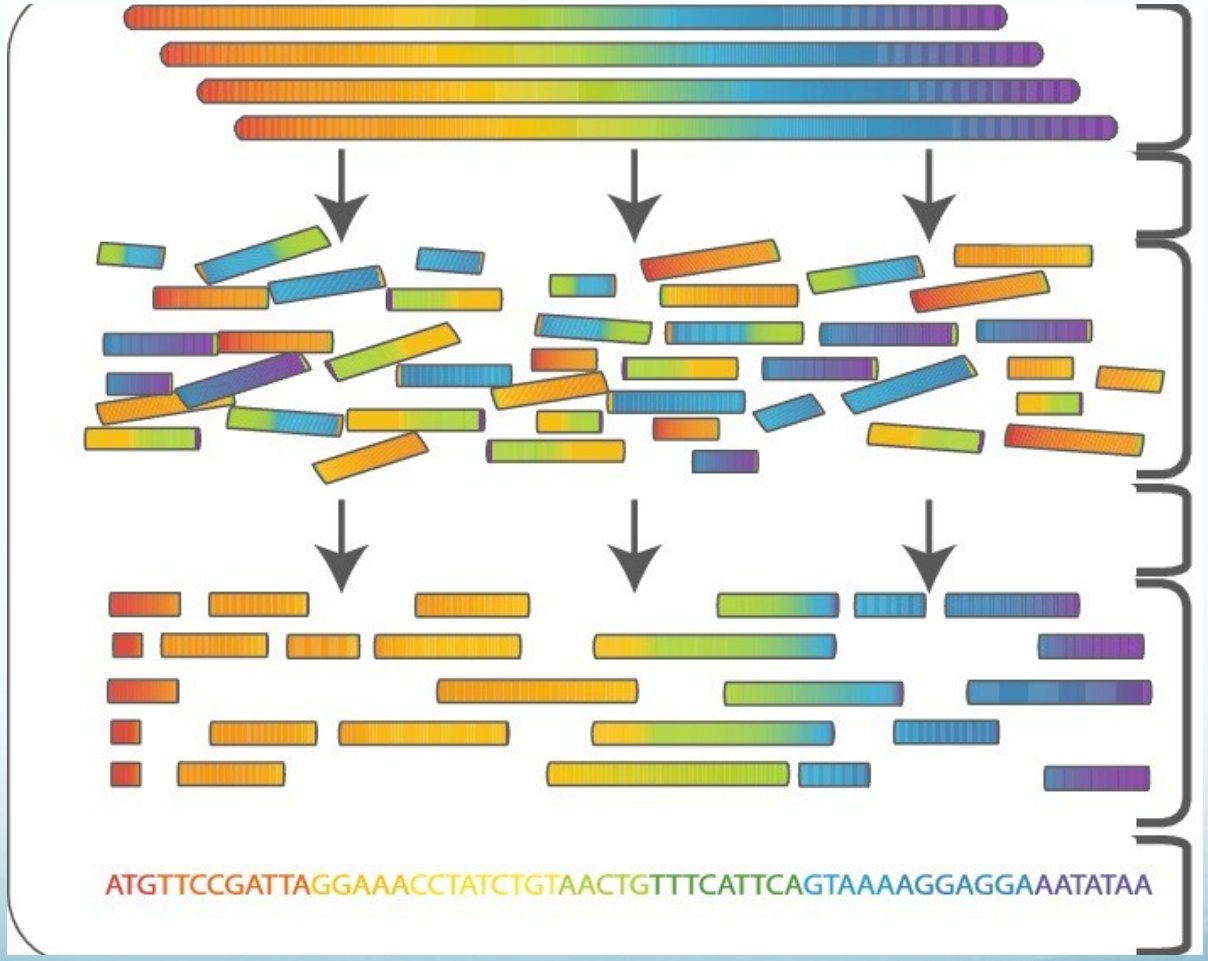
MERmaid: Distributed *de novo* Assembler

Richard Xia, Albert Kim,
Jarrod Chapman, Dan Rokhsar

The Challenge

- Historically, sequencing machines slow, but accurate
- Today, massively parallel sequencing platforms
  - Shorter reads, lower accuracy, greater redundancy
  - Total number of bases sequenced is orders of magnitude greater
- Software is needed to reassemble these short reads into a full genome

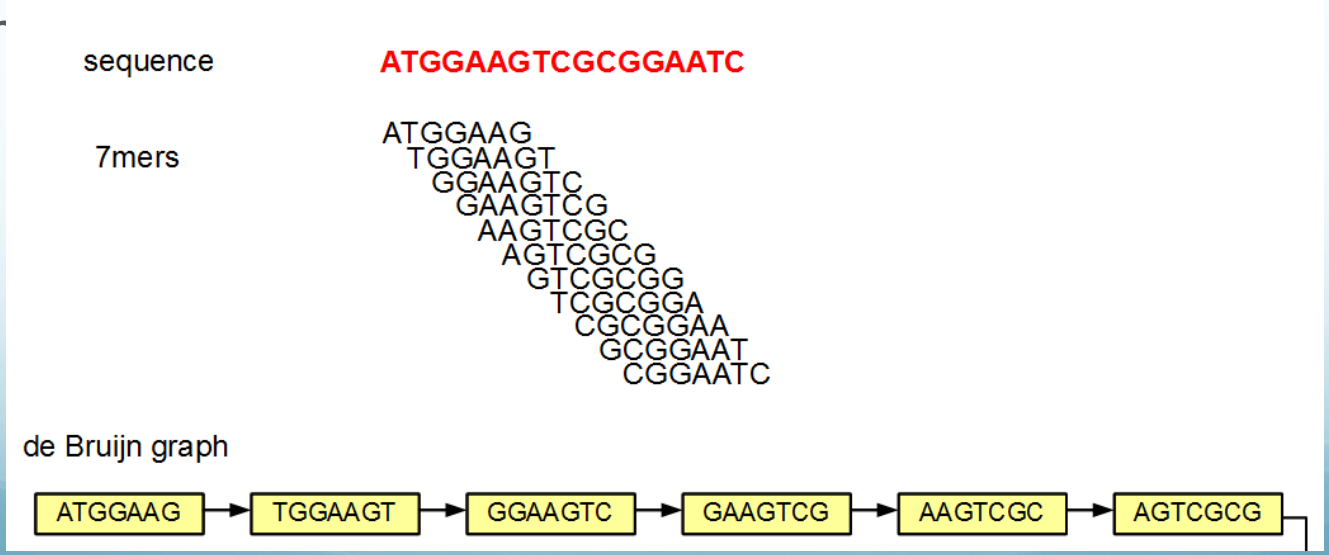
“Shotgun” Sequencing



http://www.springerimages.com/Images/Biomedicine/1-10.1007_s12575-009-9004-1-0

General Assembly (*k*-mers)

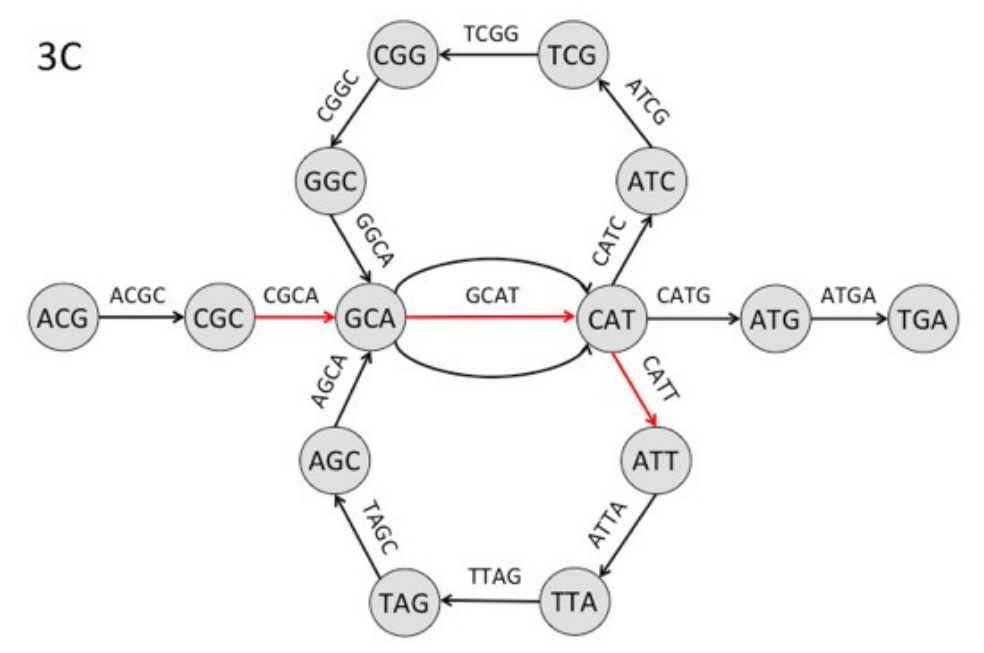
- Divide reads into smaller strings of size *k* (*k*-mers)
- Why create even smaller segments?
  - Smaller chance of containing erroneous base
  - But the tradeoff is that repetitive sequences are r



http://www.homolog.us/blogs/2011/07/28/de-bruijn-graphs-i/

General Assembly (de Bruijn Graph)

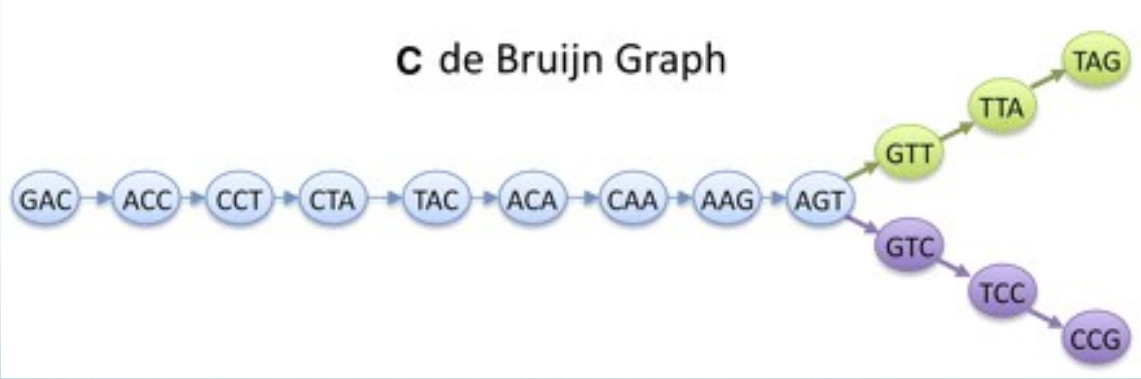
- Combine *k*-mers into a complete genome
- Model the set of *k*-mers as a de Bruijn graph
  - Nodes are *k*-mers
  - Edges are overlaps of (*k*-1) between *k*-mers



http://compbio.davidson.edu/phast/debruijn.html

General Assembly (Contigs)

- We combine **contig**uous paths of *k*-mers into longer sequences (contigs)
  - We stop when we reach a fork or a dead-end
  - We end up with the longest sequences obtained by following unambiguous edges in the de Bruijn graph

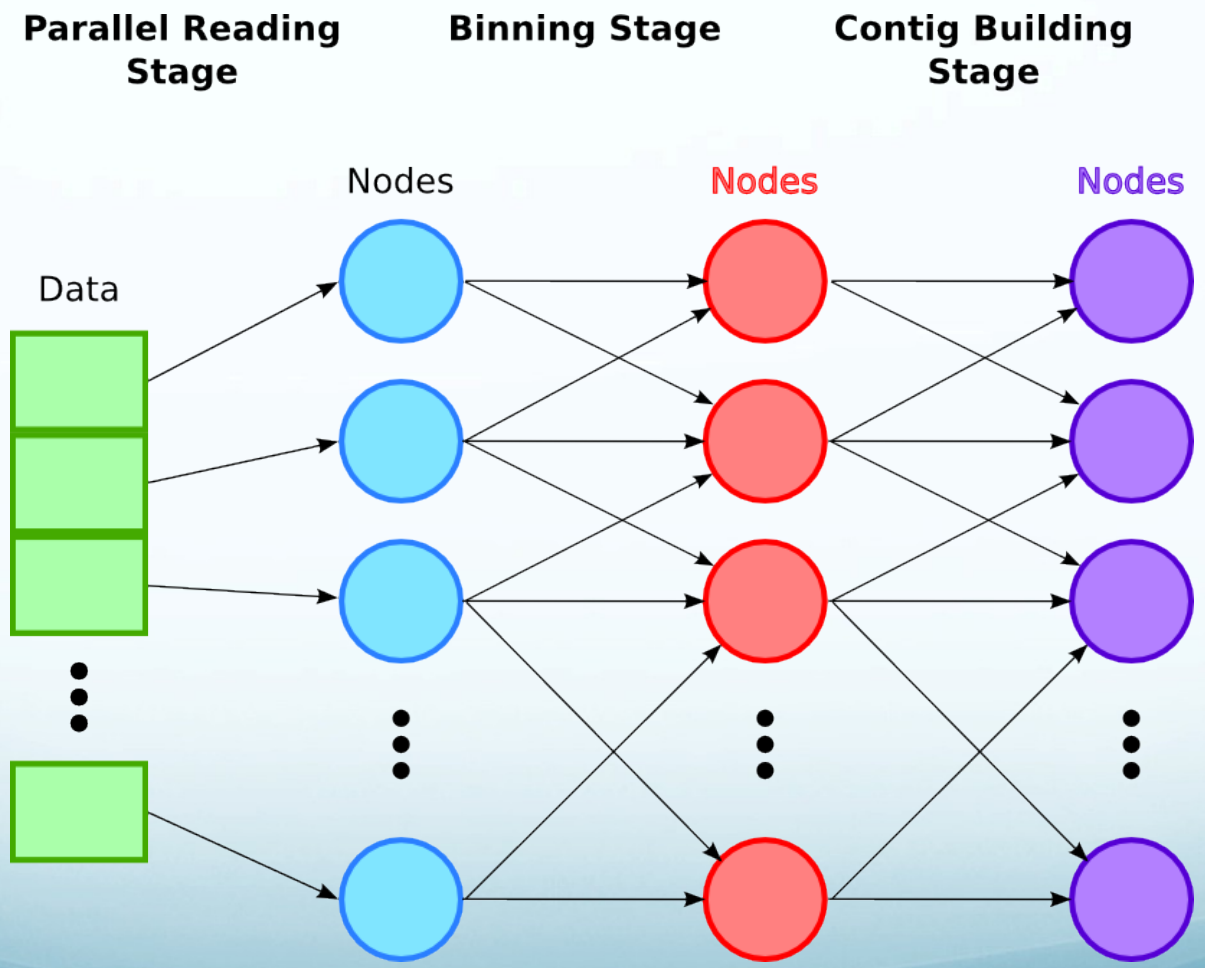


http://genome.cshlp.org/content/20/9/1165

Naïve Implementation

1. Parallel reading of data
2. Generate *k*-mers and place into bins
3. Build distributed hash table
   - Represents de Bruijn graph
   - Maps *k*-mers to their frequencies and extensions
4. Discard *k*-mers of low frequency
   - Assumed to be erroneous
5. Build contigs by walking de Bruijn graph

Architecture



Current Problems

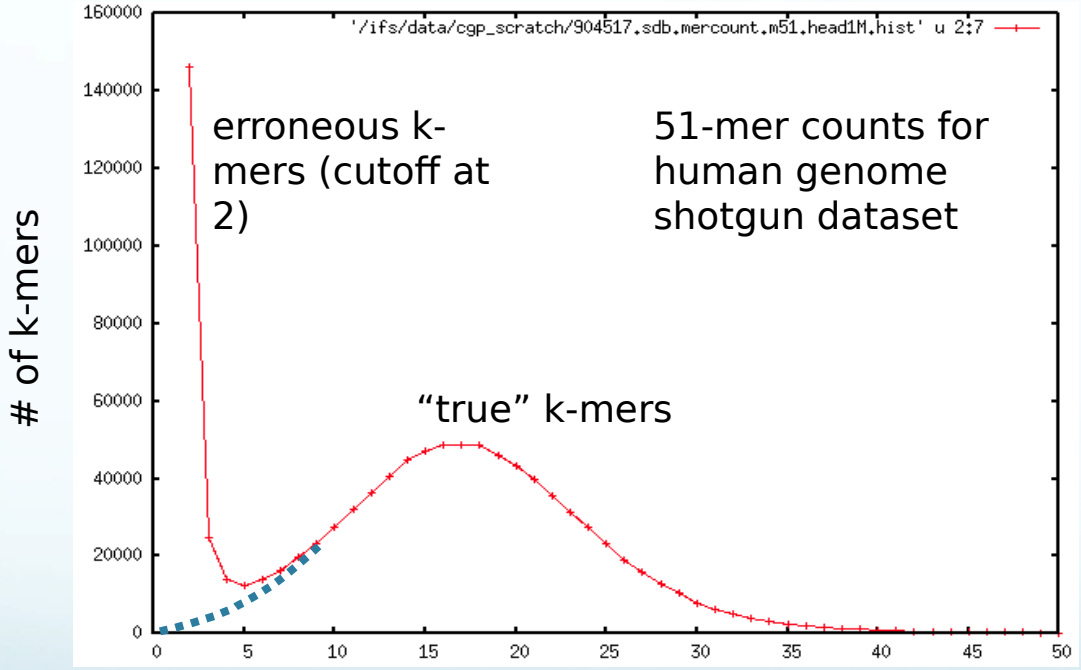
- Most current assemblers do not run on commodity clusters
  - Expected to run on supercomputers
- *k*-mer generation causes an explosion of data
  - Each read of length *l* generates (*l* – *k* + *1*) *k*-mers
- High memory usage
- I/O intensive
- Long running time

MERmaid

- Runs on commodity clusters (EC2)
- Smaller memory footprint
  - Bloom filters to exclude erroneous *k*-mers
  - Compression of *k*-mers
- Less network I/O
  - Locality-sensitive hashing

Erroneous *k*-mers

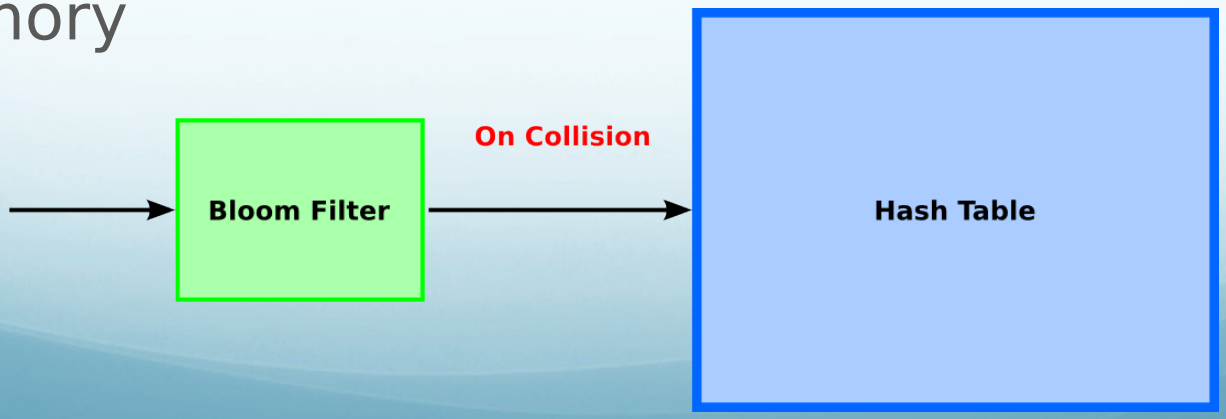
- Bi-modal distribution



http://www.plosone.org/article/info:doi%2F10.1371%2Fjournal.pone.0023501

Bloom Filters

- Use a Bloom filter
  - Used first to catch low-frequency *k*-mers
  - On collision, insert *k*-mer into traditional hash table, which saves the *k*-mer as the key
  - Erroneous *k*-mers rarely get inserted into traditional hash table
- Multiple tiers of Bloom filters to save more memory



Compression

- Transform *k*-mers to reduce encoding entropy
  - Burrows-Wheeler Transform (BWT)
  - Move-To-Front (MTF) Transform
- Run-length and Huffman Encoding
  - Binary representation dependent on compressibility of reads

Locality-Sensitive Hashing

- Bin together *k*-mers which are likely to be connected in the de Bruijn graph
- Local assembly of contigs before going off-core/off-machine
- Hash based on presence of “shingles”, or *n*-grams
- With 16 bins, 93% of extension *k*-mers can be found on same bin

Results

Number of Processes	Memory Usage	Wall Time
1	11.9 GB	51m 3s
2	12.2 GB	31m 7s
4	12.8 GB	24m 5s
8	13.6 GB	16m 53s

Pichia (a fungus)
100 Megabase Genome
No Bloom Filter, no LSH

EC2 4x Extra Large Cluster Instance
- 23 GB Memory
- 2 Quad Core Intel Xeon “Nehalem” 5570
- 10 Gigabit Ethernet

Additional Results

Type (8 processes)	Memory Usage	Wall Time
Basic	13.6 GB	16m 53s
Bloom Filter	6.28 GB	18m 23s
LSH	14.0 GB	16m 25s
Bloom Filter + LSH	6.98 GB	19m 34s

Pichia (a fungus)
100 Megabase Genome

EC2 4x Extra Large Cluster Instance
- 23 GB Memory
- 2 Quad Core Intel Xeon “Nehalem” 5570
- 10 Gigabit Ethernet

Compression

- 2-bits per base (ACGT)
  - Very difficult to compress, BWT + MTF *increase* space required by 20%
  - Theoretical limit of compression: ~20% savings
- 3-bits per base (ACGT + special characters)
  - BWT + MTF + Huffman: ~16% savings
  - BWT + Huffman: ~20% savings
- No significant savings

Conclusion/Future Work

- We did memory and I/O tradeoffs in order to make this run on commodity clusters
- 1-level Bloom filter saves ~50% memory usage
  - Future work: Experiment with multiple levels
- Compression not worthwhile
- No data to support LSH yet
  - Future work: Run on multiple nodes
- Automatic preprocessing to find cutoff parameters
- GPU implementation