Parallel Methods and Software for Next Generation Sequencing Analysis

Srinivas Aluru

School of Computational Science and Engineering

Georgia Institute of Technology

The Big Data Challenge

Then (2005)

ABI 3700

96 ~800 bp reads

76.8 X 10^3 bases

~$1 per kilo base

Now

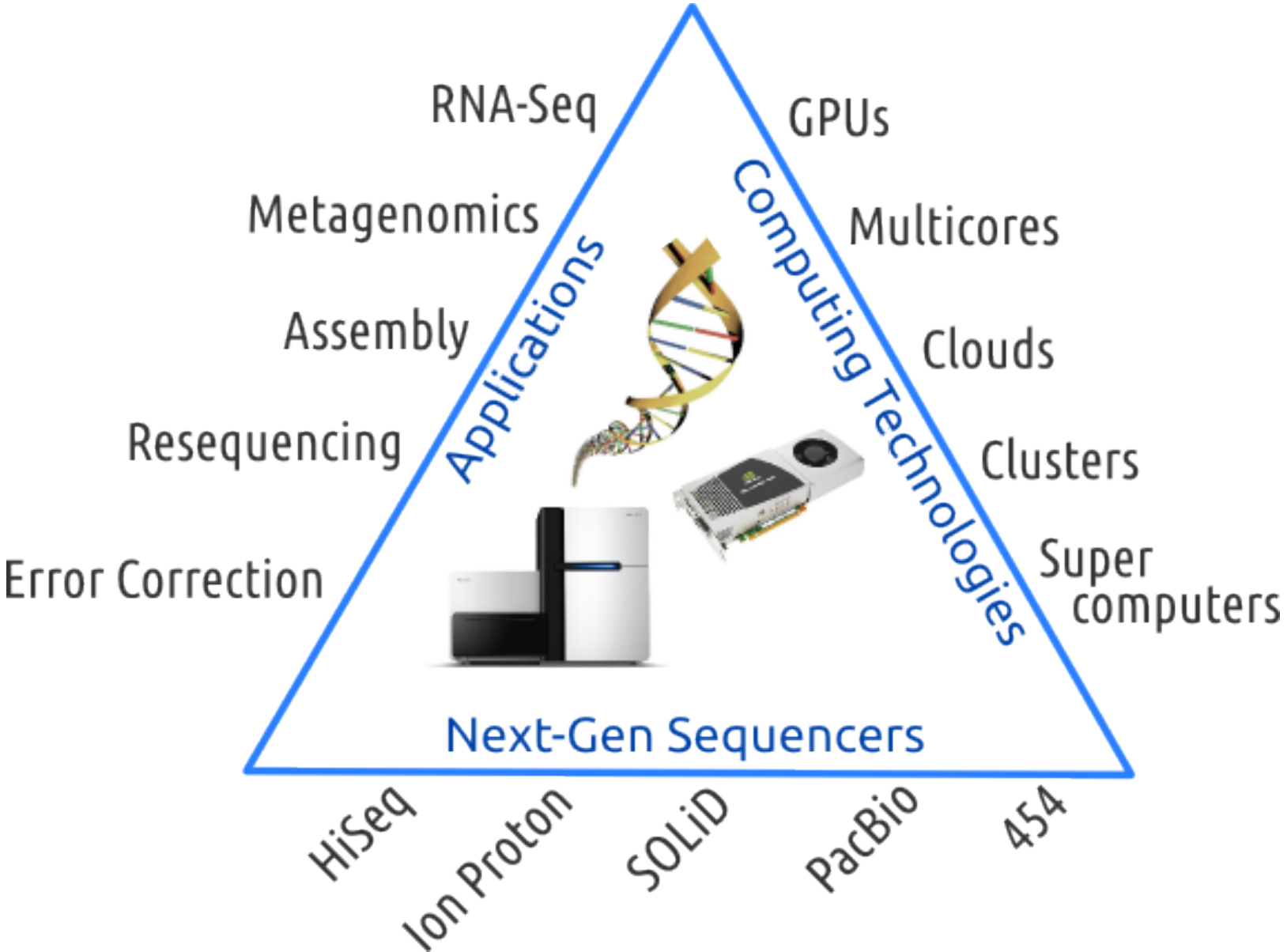
Illumina Hiseq 2500

6 billion 100 bp reads

600 X 10^9 bases

~$1 per 200 million bases

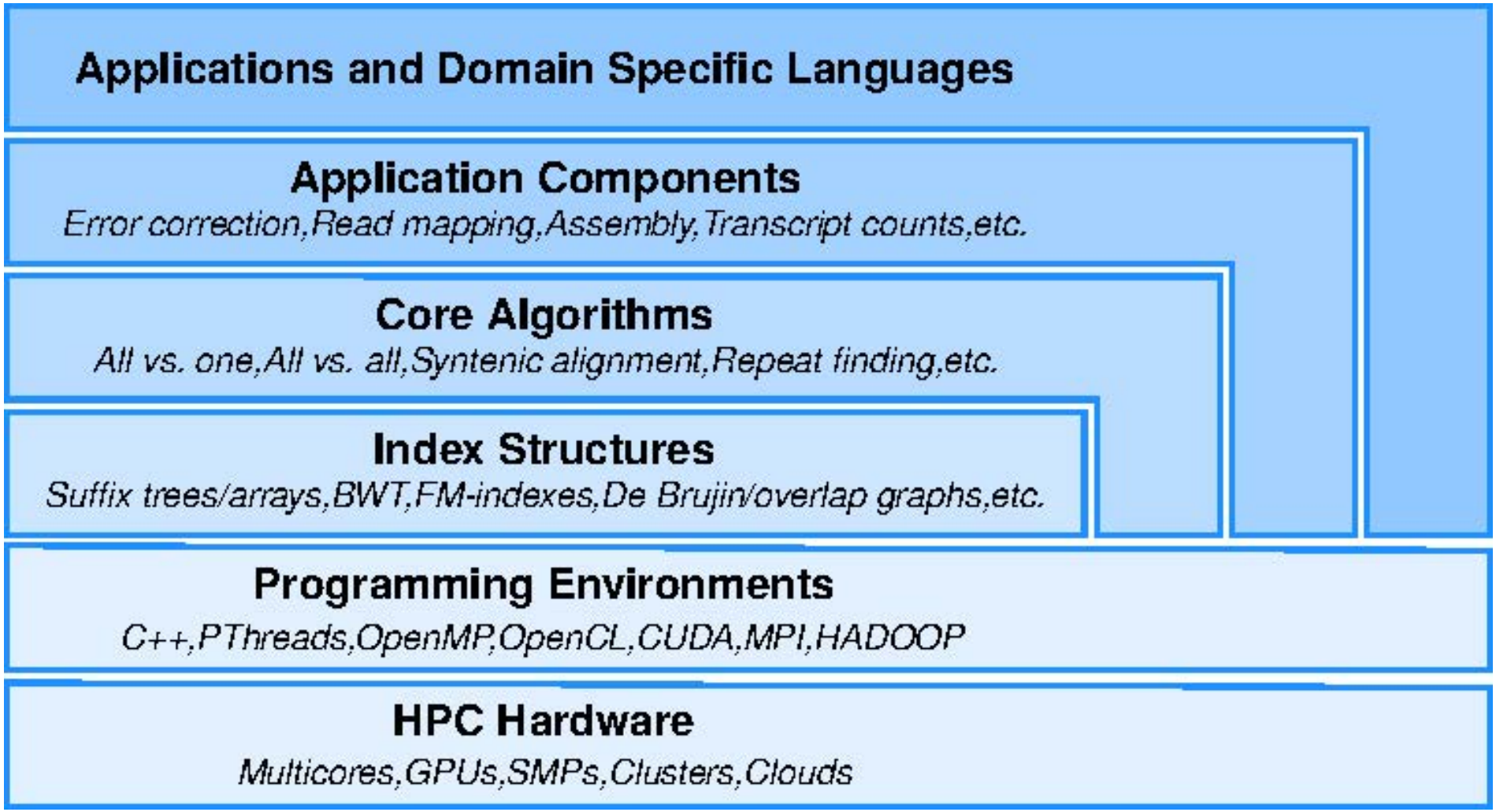
Genomes Galore – Big Data Analytics for High Throughput DNA Sequencing



Goals

- Empower bioinformatics community to migrate to HPC (and, beyond multicore shared memory)
- Preserve ability to create new bioinformatics solutions without loss of control
  - Lower levels provide higher degree of control
  - High levels reduce development time
- Should be useful both for researchers developing new methods and software developers

Software Architecture



Layer 1 – Index Structures

- Index structures on sequences
  - Look up tables, suffix trees, suffix arrays
  - BWT, FM-index
- Core data structures that operate on sequences
  - de Bruijn graphs, overlap graphs, string graphs, Hamming graphs

Layer 2 – Core Algorithms

- Support common algorithmic constructs
- Two types of sequences
  - Short (reads: 25-200bp, 450-700bp, 1000bp+)
  - Ultra-long (genomes, chromosomes, large genomic segments: 10^5 – 10^{10} bp)
- Computations within or across the two types of sequences

Layer 2 – Core Algorithms

- all vs. all between short sequences
  - Common *k*mer
  - Spaced *k*mer (used in mapping)
  - Sharing a good fraction of *k*mers (Jaccard's coeff.)
  - Local alignment (exons within genes, etc.)
  - Suffix-prefix alignment (assembly)
- all (short) vs. one (ultra-long), all vs. few
  - return all mapping locations

Layer 2 – Core Algorithms

- Intra ultra-long
  - repeats, tandem repeats, CpG islands, retrotransposons, gene duplications
- ultra-long vs. ultra-long
  - SNPs, variation detection, large-scale genomic events
- Multiple ultra-long
  - Motif detection, gene content evolution, multi-gene synteny

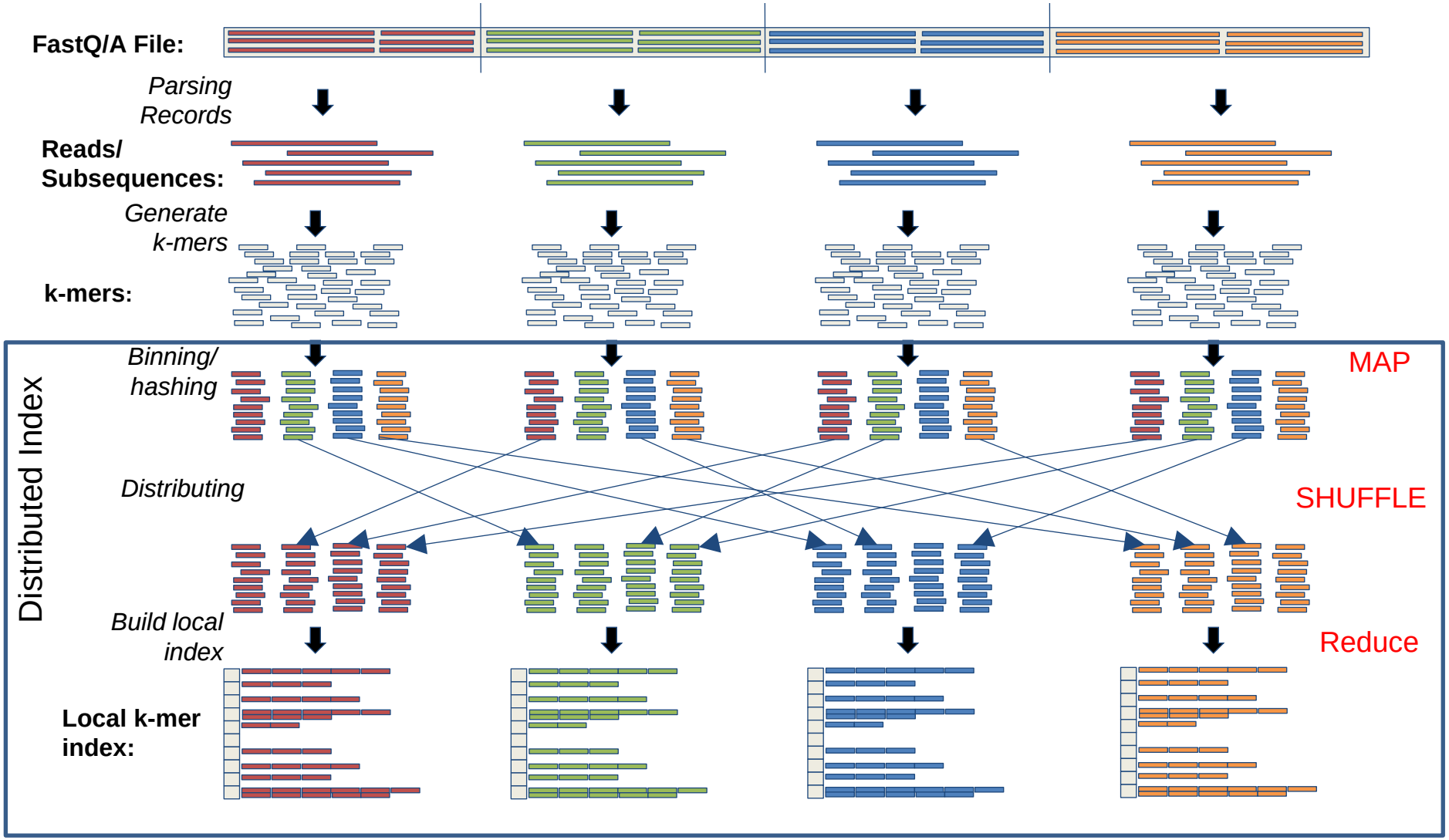
Layer 3 – Application Components

- High level processing tasks envisioned by human researchers
  - Error correction
  - Read mapping
  - Assembly
  - Transcript counts

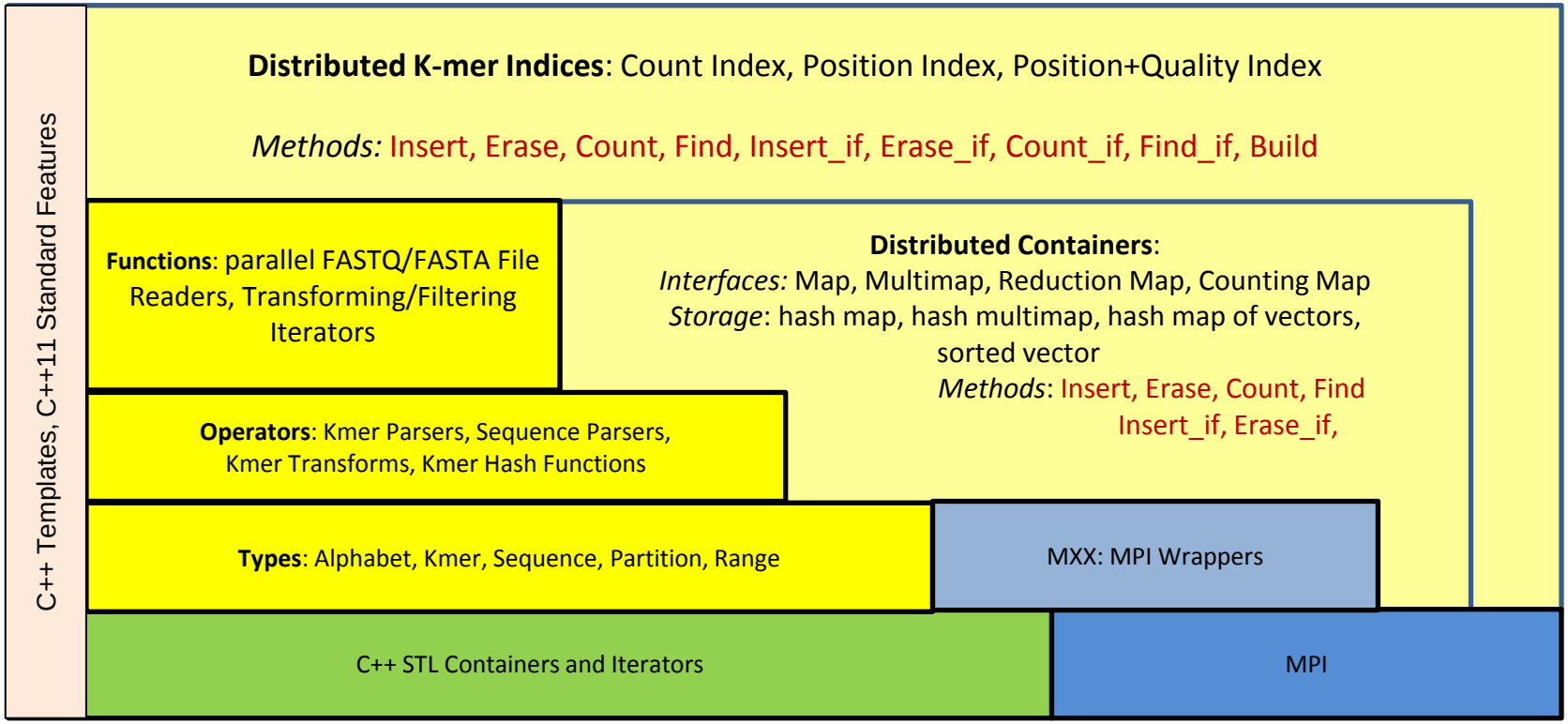
K-mer Indexing

- Create a K-mer Indexing Framework with customizable data
  - **Key:** k-mer or a transformation of a k-mer (e.g. reverse complement)
    - User choice of k, alphabet, and internal data type.
  - **Value (user specified):** read id, position in read, count/frequency, next base in read (i.e. de Bruijn graph edge), quality score, etc
- Define standard k-mer index operations
  - Provide Insert, Erase, Find, and Count operations as well as their conditional counterparts
  - Building blocks for more complex operations on k-mer indices
- Provide high performance implementations of commonly used k-mer indices
  - Indices for Count, Position, and Position and Quality Score

K-mer Indexing Algorithm



K-mer Indexing Framework Design



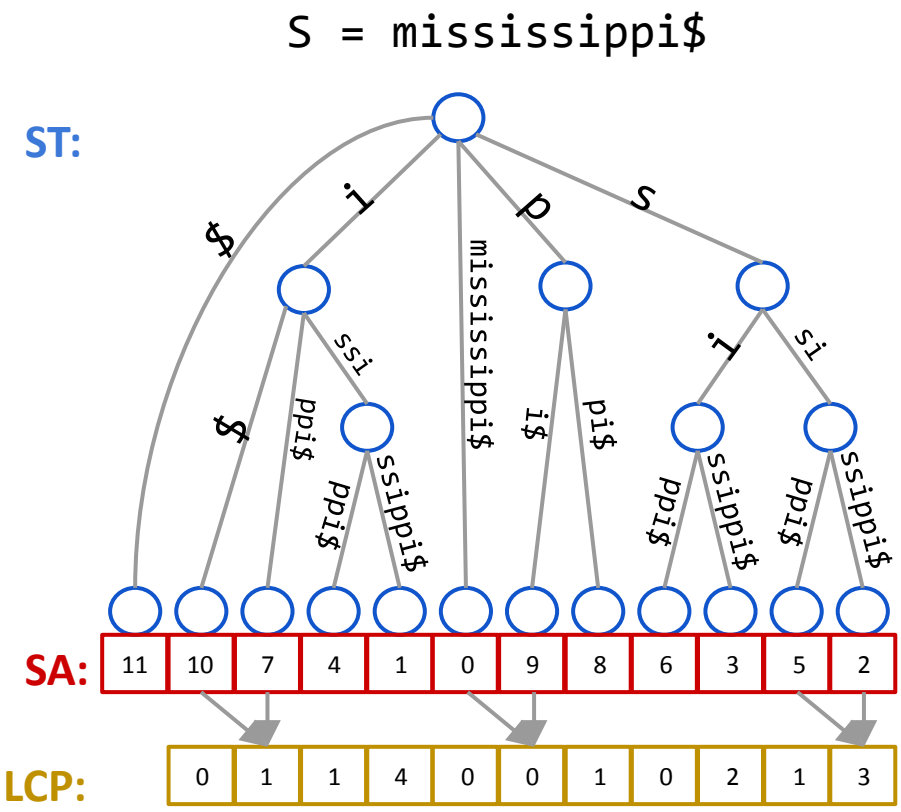
Performance Comparison

	1.9B kmers. Query with 19M kmers (2x E5-2650, 16 cores/node, passmark 15303)			1.7 B kmers, query with 9.1M kmers. Amazon EC2 instance (m2.2xlarge, Xeon X5550 4 cores, passmark 5403)							
	Count Index Hash Map of Vectors			Khmer	Jellyfish	scTurtle	KMC	DSK	Tallymer	BFCounter	Kanalyze
	32 cores	128 cores	1536 cores	8 threads, 1% false pos.	8 threads	8 threads					
Build	47.146	12.238	2.679	1500	700	500	500	1300	5400	1750	2700
Query	2.265	1.243	5.35	300	1400				1000		

- Our k-mer Indices scale to large datasets:
  - A 441GB metagenomics read set with 155.6 billion kmers
  - Count Index construction = 827sec, query for 1% kmers = 3.01 sec
- They flexibly support indices in addition to Count Index

Parallel Suffix Arrays and LCP Arrays

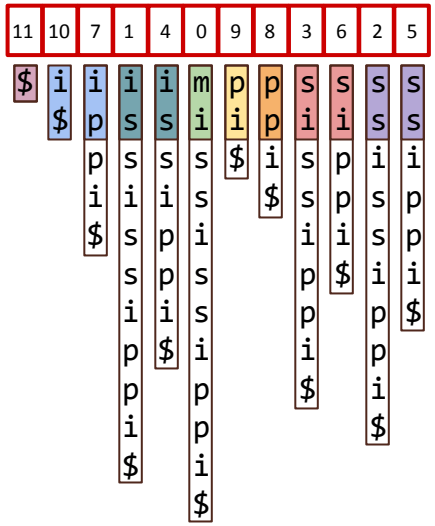
- **Suffix Tree (ST)**
  - trie of all suffixes of a string
- **Suffix Array (SA)**
  - array of sorted suffixes
  - represented by their offset
- **Longest Common Prefix (LCP)**
  - length of prefix match between consecutive suffixes in **SA**



Parallel Suffix Arrays and LCP Arrays

- **Prefix Doubling**

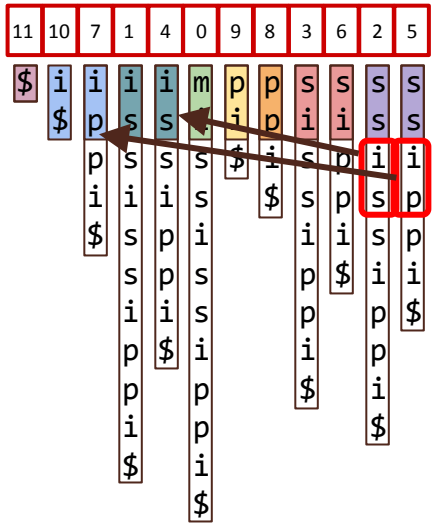
Sorted by prefix 2



Parallel Suffix Arrays and LCP Arrays

- **Prefix Doubling**

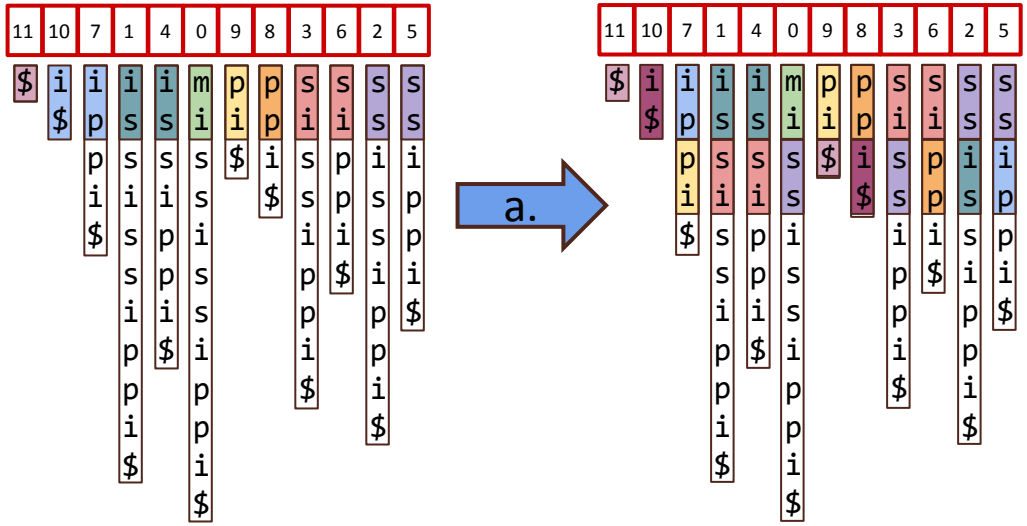
Sorted by prefix 2



Parallel Suffix Arrays and LCP Arrays

- **Prefix Doubling**

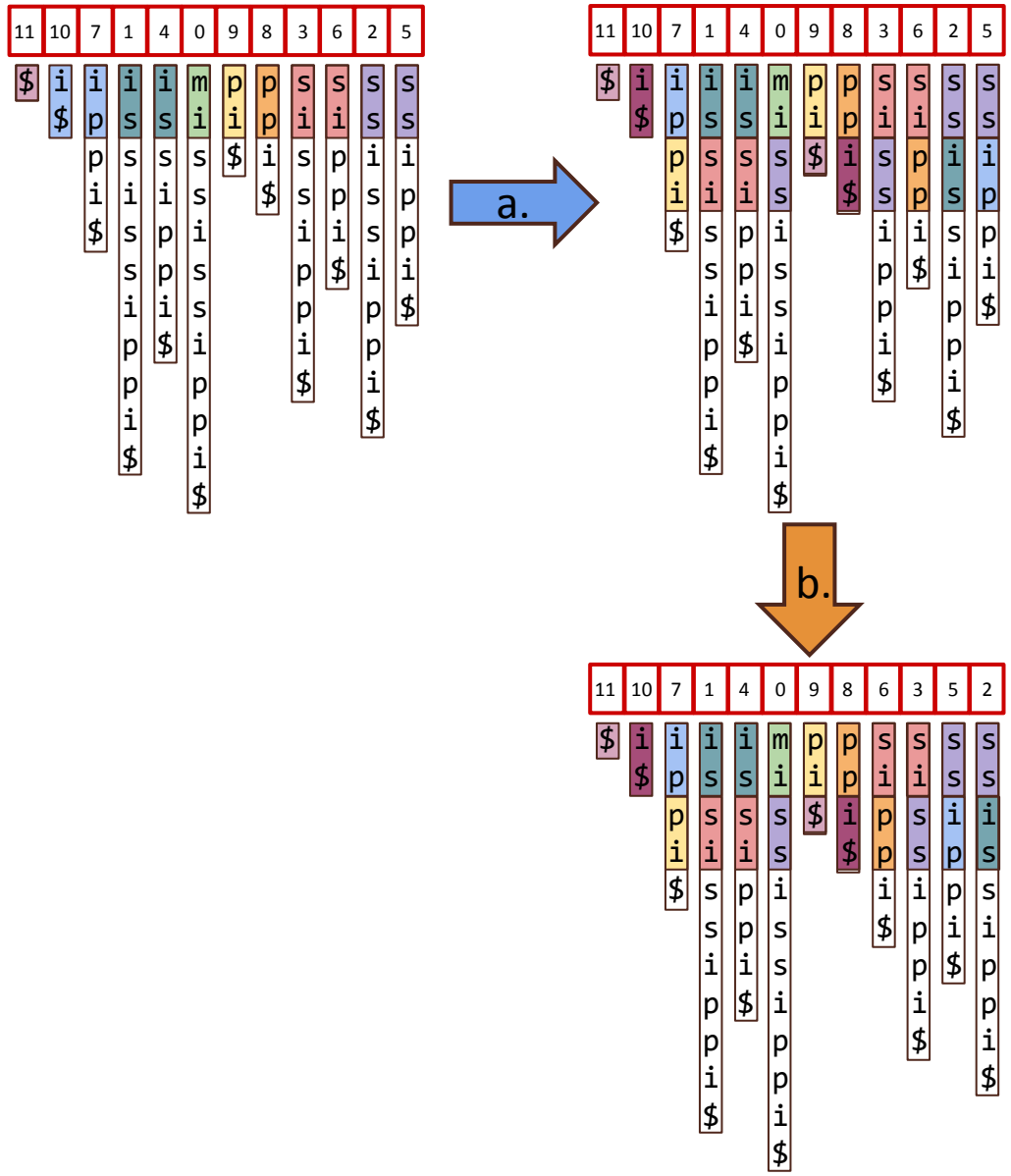
Sorted by prefix 2



Parallel Suffix Arrays and LCP Arrays

- **Prefix Doubling**

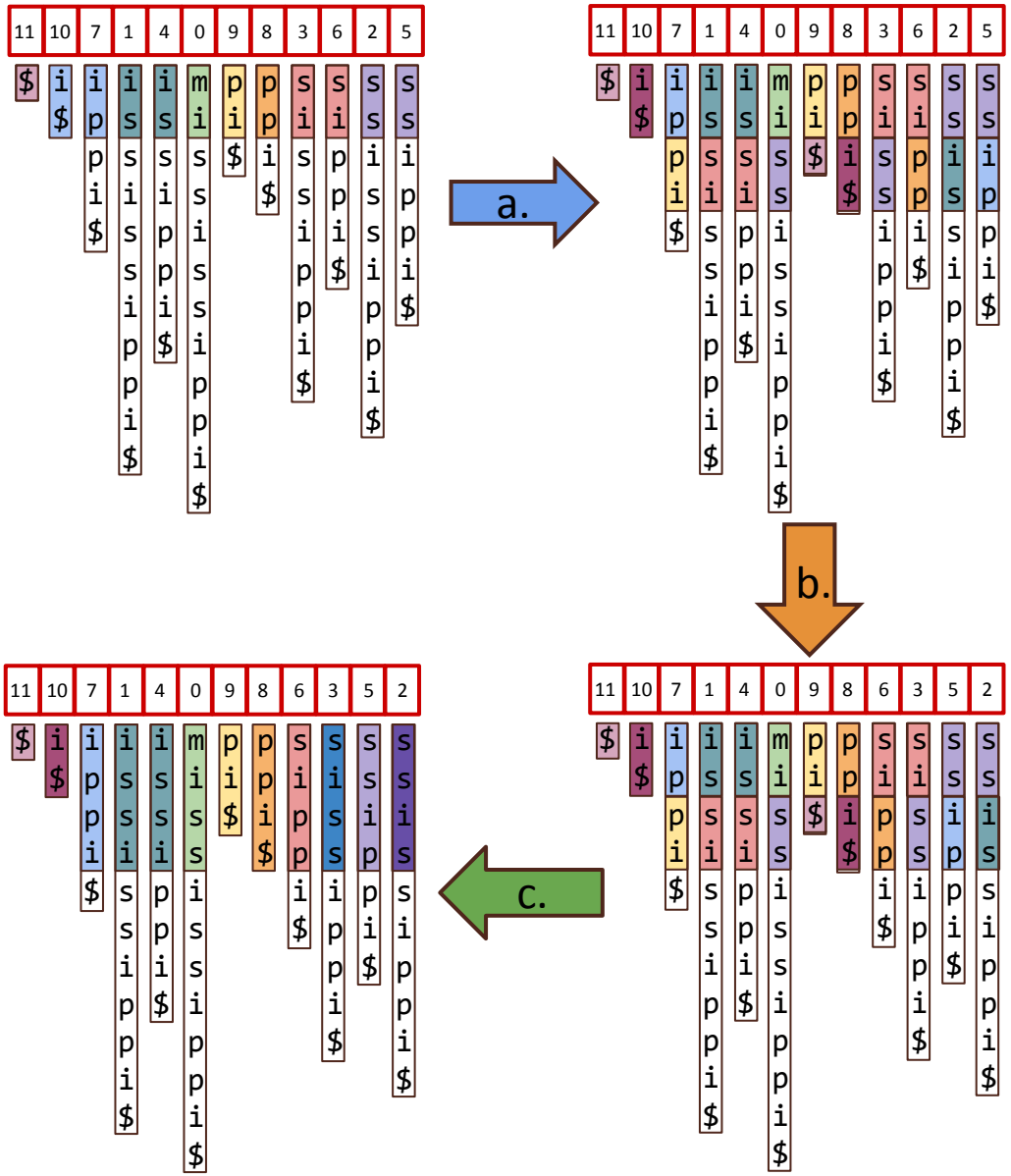
Sorted by prefix 2



Parallel Suffix Arrays and LCP Arrays

- **Prefix Doubling**

Sorted by prefix 2

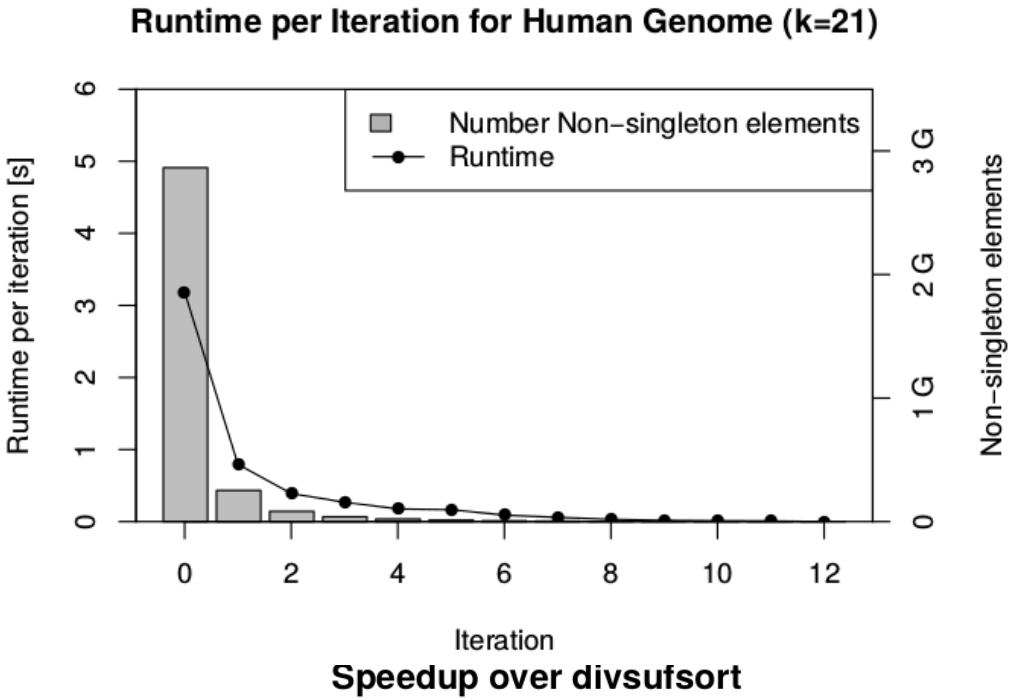


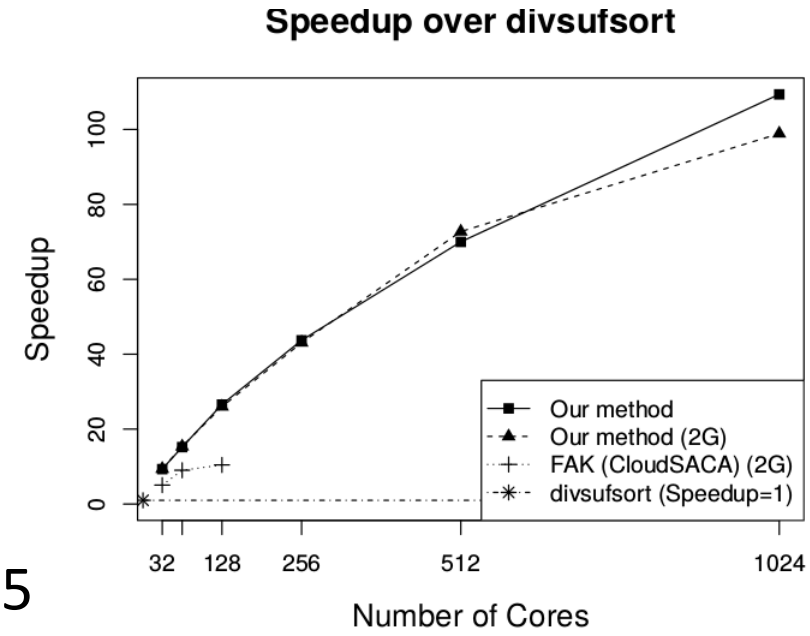
Sorted by prefix 4

Parallel Suffix Arrays and LCP Arrays

Results

- Parallel distributed-memory Suffix Array and LCP Array construction
- Scalability
  - to large inputs
  - to many nodes
- Speedup > 110x
- Indexing of full Human Genome in 8 seconds
- Outperforms all previous approaches

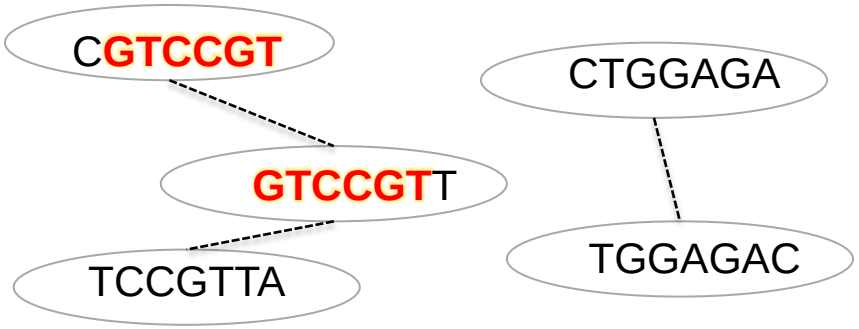




Flick and Aluru, *Supercomputing* 2015

De Bruijn Graph Partitioning

- Soil metagenomics dataset
  - Iowa Corn (1.8 billion reads)
  - Iowa Prairie (3.3 billion reads)
- High species-level heterogeneity
  - Disconnected components in de Bruijn graph (Howe *et al*. 2014)
  - 56 million components in Iowa Prairie dataset
  - 31 million components in Iowa Corn dataset

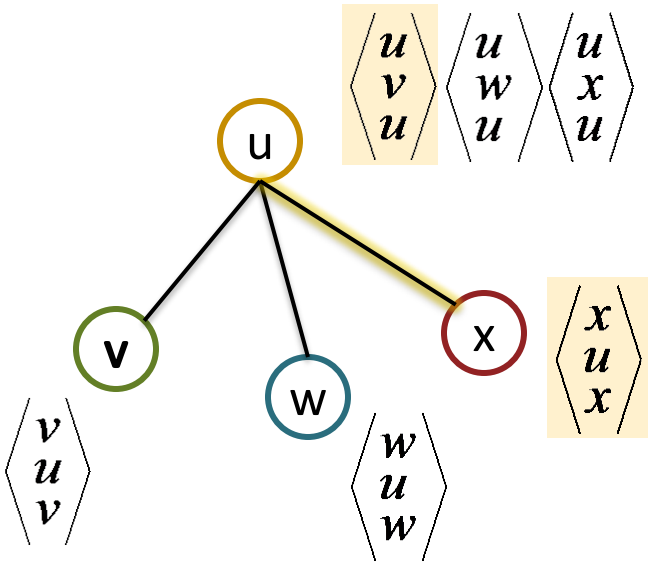


Parallel De Bruijn Graph Partitioning

- Distributed connected component labeling algorithm

Initialization

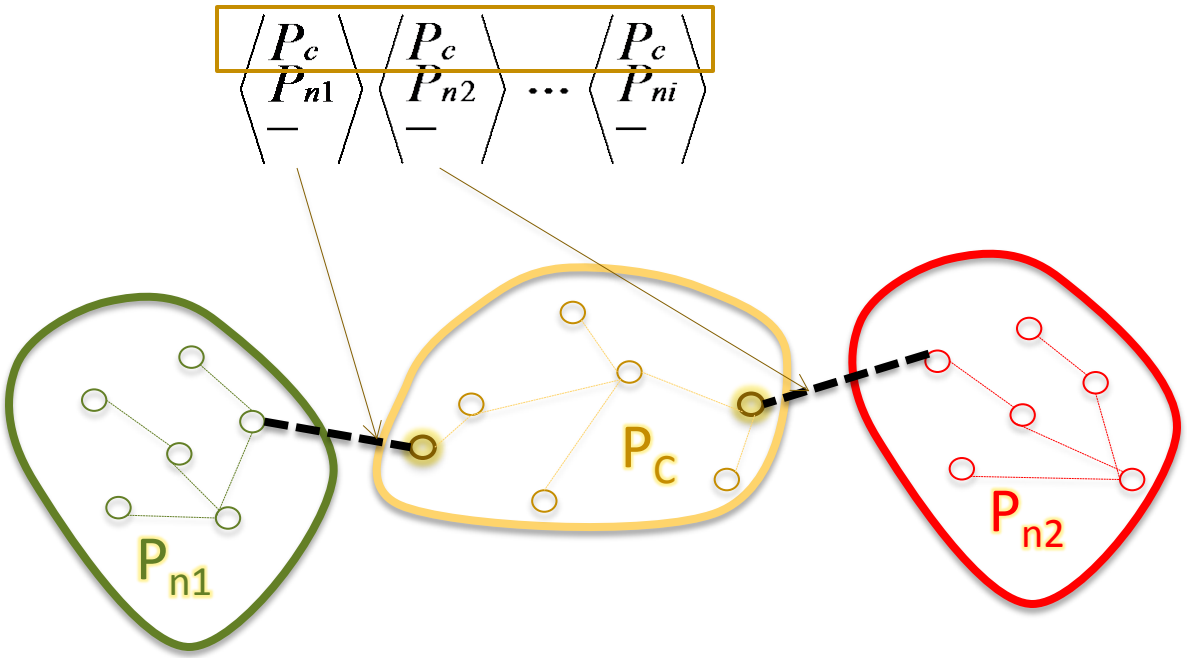
- Vector of tuples
- 2 tuples per edge in the graph
- Partition id of node = node id



⟨ Current partition id
Neighbor partition id
Node id ⟩

Parallel de Bruijn Graph Partitioning

- Each iteration
  - Do a parallel sort of all the tuples by *current partition Id*
  - Within each "bucket", compute minimum *neighbor partition id*
  - Flip the tuples to communicate my new partition id to neighbors in the next iteration

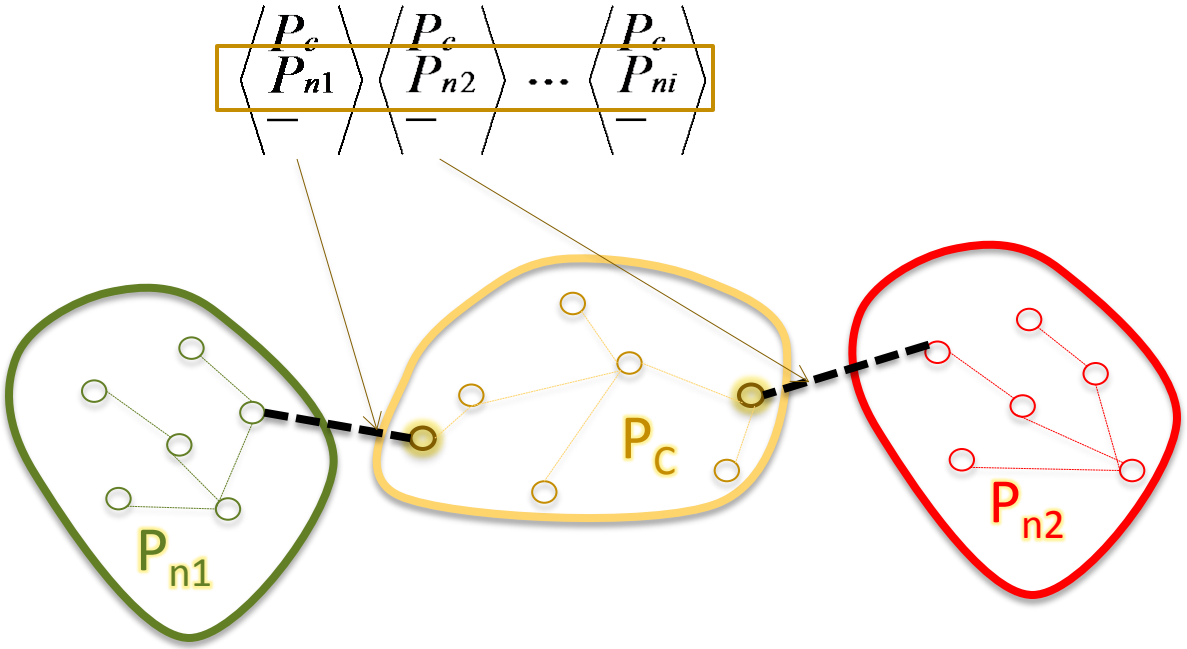


Parallel de Bruijn Graph Partitioning

- Each iteration
  - Do a parallel sort of all the tuples by *current partition Id*
  - Within each "bucket", compute minimum *neighbor partition id*

$$P_c' = \min(P_{n1}, P_{n2} \ldots P_{ni})$$

  - Flip the tuples to communicate my new partition id to neighbors in the next iteration

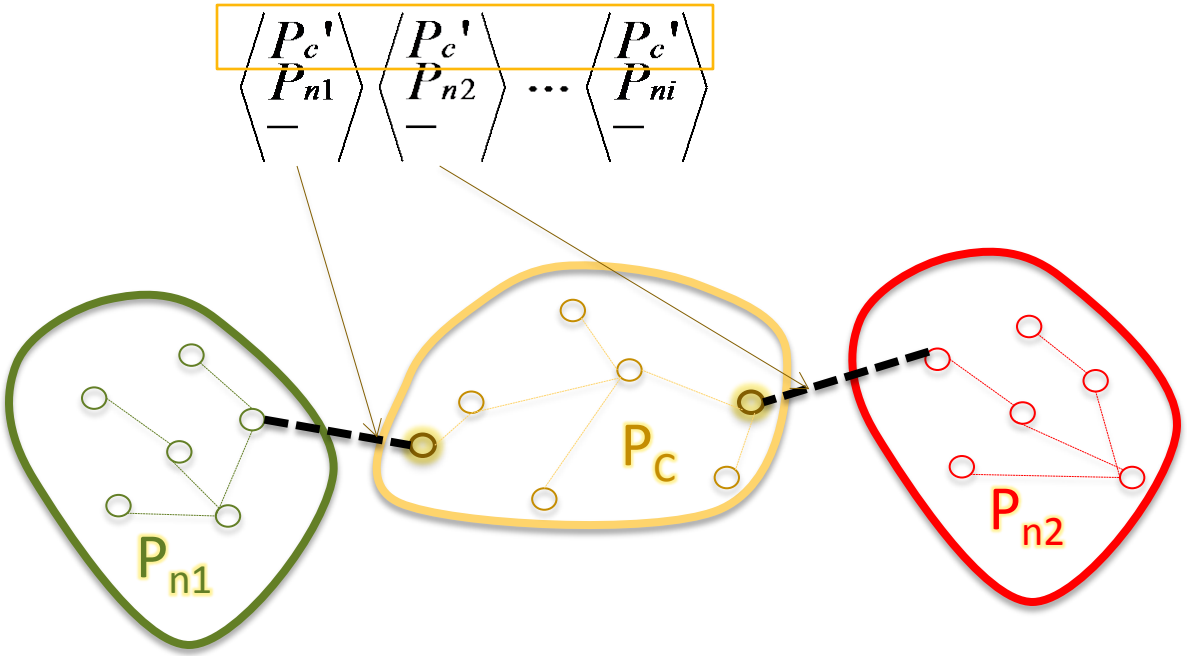


Parallel de Bruijn Graph Partitioning

- Each iteration
  - Do a parallel sort of all the tuples by *current partition Id*
  - Within each "bucket", compute minimum *neighbor partition id*

$$P_c' = \min(P_{n1}, P_{n2} \ldots P_{ni})$$
$$P_c \leftarrow P_c'$$

  - Flip the tuples to communicate my new partition id to neighbors in the next iteration

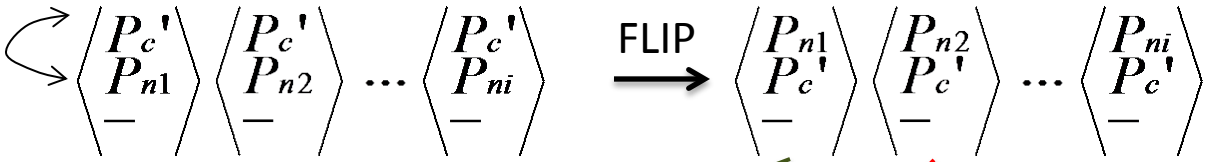


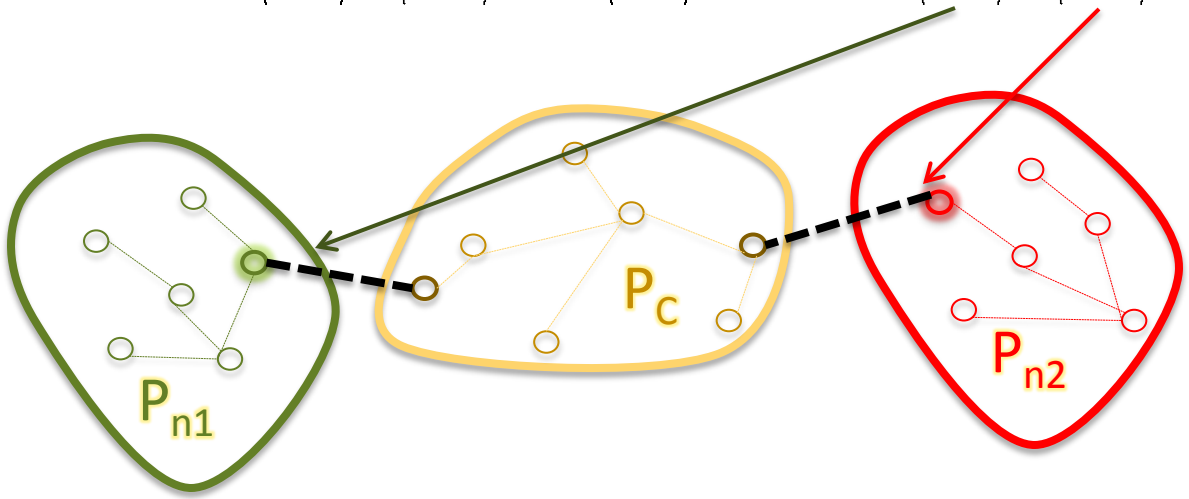
Parallel de Bruijn Graph Partitioning

- Each iteration
  - Do a parallel sort of all the tuples by *current partition Id*
  - Within each "bucket", compute minimum *neighbor partition id*

$$P_c' = \min(P_{n1}, P_{n2} \ldots P_{ni})$$
$$P_c \leftarrow P_c'$$

  - Flip the tuples to communicate my new partition id to neighbors in the next iteration





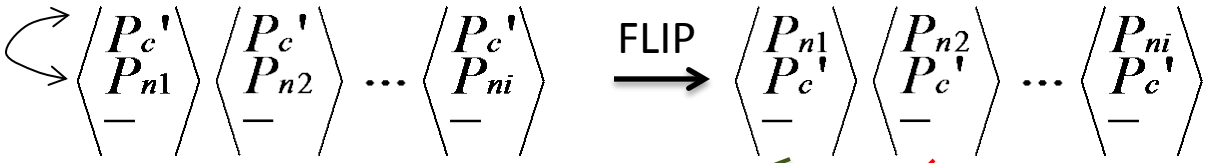
Parallel de Bruijn Graph Partitioning

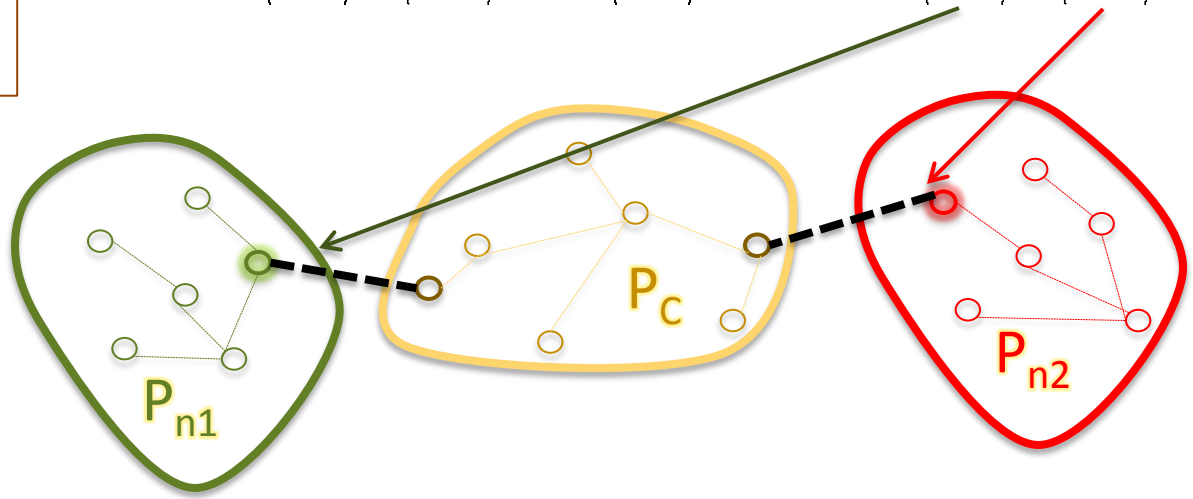
- Each iteration
  - Do a parallel sort of all the tuples by *current partition Id*
  - Within each "bucket", compute minimum *neighbor partition id*

$$P_c' = \min(P_{n1}, P_{n2} \ldots P_{ni})$$
$$P_c \leftarrow P_c'$$

  - Flip the tuples to communicate my new partition id to neighbors in the next iteration

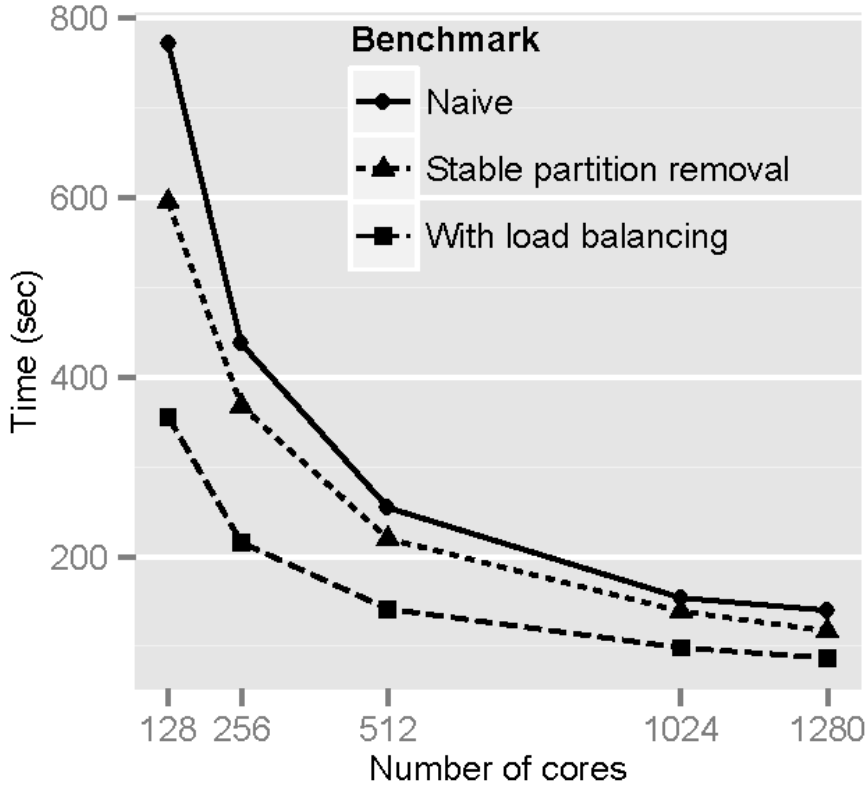
Loop until convergence





Parallel de Bruijn Graph Partitioning

Strong scalability upto 1280 Xeon cores using Infiniband.



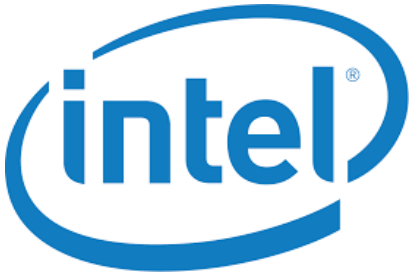
- Partitioned graph with 135 billion edges in 22 minutes (for Iowa corn metagenomics dataset)
- Sequential method takes multiple days.
- Prior methods for connected component labeling don't scale beyond 40 cores on sparse graphs.

Flick, Jain, Pan, and Aluru, *Supercomputing* 2015

Acknowledgements

Research Group:

- Patrick Flick
- Chirag Jain
- Yongchao Liu
- Tony Pan
- Support provided by:



Collaborators:

- Jaroslaw Zola, SUNY Buffalo
- Patrick Schnable, Iowa State
- Charles Sing, U. of Michigan
- Kunle Olukotun, Stanford
- Wu-Chun Feng, V. Tech