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 $10^3$ bases
~$1$ per kilo base

Now

Illumina Hiseq 2500
6 billion 100 bp reads
600 $10^9$ bases
~$1$ per 200 million bases
Genomes Galore – Big Data Analytics for High Throughput DNA Sequencing

Applications:
- RNA-Seq
- Metagenomics
- Assembly
- Resequencing
- Error Correction

Computing Technologies:
- GPUs
- Multicores
- Clouds
- Clusters
- Super computers

Next-Gen Sequencers:
- HiSeq
- Ion Proton
- SOLiD
- PacBio
- 454
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

Applications and Domain Specific Languages

Application Components
Error correction, Read mapping, Assembly, Transcript counts, etc.

Core Algorithms
All vs. one, All vs. all, Syntenic alignment, Repeat finding, etc.

Index Structures
Suffix trees/arrays, BWT, FM-indexes, De Bruijn/overlapping graphs, etc.

Programming Environments
C++, PThreads, OpenMP, OpenCL, CUDA, MPI, HADOOP

HPC Hardware
Multicores, GPUs, SMPs, Clusters, Clouds
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

FastQ/A File: 

Parsing Records

Reads/Subsequences:

Generate k-mers

k-mers:

Binning/hashing

Distributing k-mers:

Build local index

Local k-mer index:

Distributed Index

MAP

SHUFFLE

Reduce
K-mer Indexing Framework Design

**Distributed K-mer Indices**: Count Index, Position Index, Position+Quality Index

**Methods**: Insert, Erase, Count, Find, Insert_if, Erase_if, Count_if, Find_if, Build

**Distributed Containers**:
- **Interfaces**: Map, Multimap, Reduction Map, Counting Map
- **Storage**: hash map, hash multimap, hash map of vectors, sorted vector
- **Methods**: Insert, Erase, Count, Find, Insert_if, Erase_if,

**Functions**: parallel FASTQ/FASTA File Readers, Transforming/Filtering Iterators

**Operators**: Kmer Parsers, Sequence Parsers, Kmer Transforms, Kmer Hash Functions

**Types**: Alphabet, Kmer, Sequence, Partition, Range

**C++ STL Containers and Iterators**

**MXX**: MPI Wrappers

**MPI**
## Performance Comparison

<table>
<thead>
<tr>
<th>Count Index Hash Map of Vectors</th>
<th>32 cores</th>
<th>128 cores</th>
<th>1536 cores</th>
<th>8 threads, 1% false pos.</th>
<th>8 threads</th>
<th>8 threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>Build</td>
<td>47.146</td>
<td>12.238</td>
<td>2.679</td>
<td>1500</td>
<td>700</td>
<td>500</td>
</tr>
<tr>
<td>Query</td>
<td>2.265</td>
<td>1.243</td>
<td>5.35</td>
<td>300</td>
<td>1400</td>
<td>1000</td>
</tr>
</tbody>
</table>

### Performance Data
- **Our k-mer Indices scale to large datasets:**
  - A 441GB metagenomics read set with 155.6 billion kmers
  - Count Index construction = 827 sec, 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

\[ S = \text{mississippi}\$

\begin{align*}
\text{ST:} & \\
\text{SA:} & 11 \ 10 \ 7 \ 4 \ 1 \ 0 \ 9 \ 8 \ 6 \ 3 \ 5 \ 2 \\
\text{LCP:} & 0 \ 1 \ 1 \ 4 \ 0 \ 0 \ 1 \ 0 \ 2 \ 1 \ 3
\end{align*}
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

\[
\begin{array}{c}
\text{Current partition id} \\
\text{Neighbor partition id} \\
\text{Node id}
\end{array}
\]

\[
\begin{array}{c}
\langle u, v \rangle \\
\langle u, w \rangle \\
\langle u, x \rangle
\end{array}
\]

\[
\begin{array}{c}
\langle v, u \rangle \\
\langle v, w \rangle
\end{array}
\]

\[
\begin{array}{c}
\langle w, u \rangle \\
\langle w, w \rangle
\end{array}
\]
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
• 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
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
• Each iteration
  – Do a parallel sort of all the tuples by \textit{current partition id}
  – Within each “bucket”, compute minimum \textit{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

\[ \left\langle \frac{P_{c}'}{P_{n1}} \right\rangle, \left\langle \frac{P_{c}'}{P_{n2}} \right\rangle, \ldots, \left\langle \frac{P_{c}'}{P_{ni}} \right\rangle \xrightarrow{\text{FLIP}} \left\langle \frac{P_{n1}}{P_{c}'} \right\rangle, \left\langle \frac{P_{n2}}{P_{c}'} \right\rangle, \ldots, \left\langle \frac{P_{ni}}{P_{c}'} \right\rangle \]
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**

\[
P_c' = \min(P_{n1}, P_{n2}...P_{ni})
\]

\[
P_c \leftarrow P_c'
\]
- 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