Scalable Parallel Algorithms for De Novo Assembly of Complex Genomes

Evangelos Georganas$^{1,2}$

with $^2$Aydın Buluç, $^3$Jarrod Chapman, $^2$Steven Hofmeyr, $^1$Chaitanya Aluru, $^3$Rob Egan, $^2$Leonid Oliker, $^3$Daniel Rokhsar, $^{1,2}$Katherine Yelick

$^1$EECS Department, University of California, Berkeley
$^2$Computational Research Division, Lawrence Berkeley National Laboratory
$^3$Joint Genome Institute, Lawrence Berkeley National Laboratory
Outline

• The problem of genome assembly

• Distributed de Bruijn graphs

• Parallel de Bruijn graph construction & traversal

• Performance results

• Summary
**De novo Genomes Assembly**

- **DNA sequence consists of 4 bases:** A/C/G/T

- **Read:** short fragment of DNA sequence that can be read by a DNA sequencing technology – can’t read whole DNA in one go.

- **De novo genome assembly:** Reconstruct an unknown genome from a collection of short reads.
  - Constructing a jigsaw puzzle without seeing the picture on the box
## Genomes vary in size

<table>
<thead>
<tr>
<th>Organism</th>
<th>Genome size (in billion bases)</th>
<th>Typical read data size</th>
</tr>
</thead>
<tbody>
<tr>
<td>E. Coli</td>
<td>0.5</td>
<td>3 Gb</td>
</tr>
<tr>
<td>Hagfish</td>
<td>1.5</td>
<td>300 Gb</td>
</tr>
<tr>
<td>Human</td>
<td>3</td>
<td>650 Gb</td>
</tr>
<tr>
<td>Wheat</td>
<td>17</td>
<td>1200 Gb</td>
</tr>
<tr>
<td>Salamander</td>
<td>20</td>
<td>1400 Gb</td>
</tr>
</tbody>
</table>
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 ~100 (Illumina) to 30K (PacBio) bases.
  - Genomes vary in length and complexity – up to 30G bases.

- Reads include errors.

- Genomes have repetitive regions.
  - Repetitive regions increase genome complexity.
  - Analogy: Same pieces of a puzzle appear in multiple places.
De novo genome assembly is expensive!

- Human genome (3 Gbp) “de novo” assembled:
  - SGA assembler: 140 hours
  - Meraculous assembler: 48 hours

- Wheat genome (17 Gbp) “de novo” assembled (2014):
  - Meraculous assembler: 170 hours (projected)
  - Would require a machine with 512 GB RAM

- Pine genome (20 Gbp) “de novo” assembled (2014):
  - Masurca assembler: 3 months on a machine with 1 TB RAM
De novo Genome Assembly a la Meraculous

**Input:** Reads that may contain errors

1. Chop reads into k-mers, process k-mers to exclude errors
   - Intensive I/O
   - High memory footprint
   - High bandwidth requirement

2. Construct & traverse de Bruijn graph of k-mers, generate contigs
   - Huge graph as a hash table
   - Irregular accesses
   - Injection limited

3. Leverage read information to link contigs and generate scaffolds.
   - High memory requirements
   - Intensive computation
   - Intensive I/O
De novo Genome Assembly a la Meraculous

List of publications

1. Parallel De Bruijn Graph Construction and Traversal for de novo Genome Assembly, SC’14.
   E.G., Aydin Buluç, Jarrod Chapman, Leonid Oliker, Daniel Rokhsar, Katherine Yelick

2. merAligner: A Fully Parallel Sequence Aligner, IPDPS’15.
   E.G., Aydin Buluç, Jarrod Chapman, Leonid Oliker, Daniel Rokhsar, Katherine Yelick

3. HipMer: An Extreme-Scale De Novo Genome Assembler, SC’15.
   E.G., Aydin Buluç, Jarrod Chapman, Steven Hofmeyr, Chaitanya Aluru, Rob Egan, Leonid Oliker, Daniel Rokhsar, Katherine Yelick

In this talk: de Bruijn graph construction and traversal
Insights for leveraging Unified Parallel C

- Unified Parallel C (UPC) is a Partitioned Global Address Space language with one-sided communication.

- Core graph algorithms implemented in UPC where graph $\Leftrightarrow$ hash table.

- We need a huge distributed hash table:
  - **Key idea 1:** Use aggregate distributed memory to meet the demands!

- Irregular access pattern in the hash table:
  - **Key idea 2:** One-sided communication.

- Result of this work: *Complete assembly of human genome in 8 minutes using 15K cores*

- Original code required **2 days** and a large memory machine (500 GB RAM)

- Portable implementation with UPC: runs on any machine (your laptop, your cluster, your supercomputer) without any change!
Distributed De Bruijn Graph

- The de Bruijn graph of k-mers is represented as a hash table.
- A k-mer is a node in a graph $\Leftrightarrow$ a k-mer is an entry (key) in the hash table.
- An edge in the graph connects two nodes that overlap in k-1 bases.
- The edges in the hash table can be stored efficiently by storing the extensions of the k-mers as their corresponding values.
- The connected components represent *contigs*.

Contig 1: GATCTGA

Contig 2: AACCG

Contig 3: AATGC
Parallel De Bruijn Graph Construction

**Input:** k-mers and their high quality extensions

**Read k-mers & extensions**

**Store k-mers & extensions**

**Distributed Hash table**

- **Shared Hash table**
  - **buckets**
  - **entries**
  - **Key:** ATC **Val:** TG  **Key:** ACC **Val:** GA
  - **Key:** AAC **Val:** CF
  - **Key:** TGA **Val:** FC
  - **Key:** GAT **Val:** CF
  - **Key:** AAT **Val:** GF
  - **Key:** TCT **Val:** GA
  - **Key:** CCG **Val:** FA
  - **Key:** CTG **Val:** AT
  - **Key:** TGC **Val:** FA

- **Private Hash table**
  - **buckets**
  - **entries**
  - **Key:** TGC **Val:** FA
  - **Key:** ATG **Val:** CA
  - **Key:** GAT **Val:** CF
  - **Key:** TCT **Val:** GA
  - **Key:** ATG **Val:** CA
  - **Key:** TCT **Val:** GA
  - **Key:** CTG **Val:** AT
  - **Key:** TGC **Val:** FA

- **Global Address Space**
  - **x**, **y**, **z**

- **Fine-grained communication & fine-grained locking required**

---

**Tuesday, April 12th, 2016**

**SIAM PP16**
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 Construction

- **Challenge 1:** The hash table that represents the de Bruijn graph is huge (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 table construction introduces communication and synchronization costs
  - **Solution:** We can split the construction in two phases and aggregate messages to reduce number of messages and synchronization $\rightarrow$ 10x-20x performance improvement.
Algorithm: Pick a random traversal seed and expand connected component by consecutive lookups in the distributed hash table.
Parallel De Bruijn Graph Traversal

- **Algorithm**: Pick a random traversal seed and expand connected component by consecutive lookups in the distributed hash table.
- **Fine-grained, irregular, remote accesses.** Need fine-grained parallelization.
  - To the extreme, the result can be a single very long chain (high-diameter graph).
  - Global address space and one-sided communication of UPC simplifies logic.
- If multiple processors are working on the same connected component, they cooperate via a lightweight synchronization protocol.
A processor picks a k-mer as seed and initiates a subcontig.

**Lightweight synchronization protocol**

- **ACTIVE**
  - Found USED k-mers in both directions of the subcontig AND one of the neighboring subcontigs is ABORTED:
    1) Attach that subcontig to local subcontig,
    2) Set that state to ACTIVE
  - Reached both endpoints of a UU contig.
    1) Set state to COMPLETE and store UU contig
    2) Pick another random seed from hash table

- **INACTIVE**
  - A processor picks a k-mer as seed and initiates a subcontig.

- **ABORTED**
  - Attached in a neighboring subcontig

- **COMPLETE**
  - Reached both endpoints of a UU contig.
    1) Set state to COMPLETE and store UU contig
    2) Pick another random seed from hash table

- Found UNUSED k-mer in any direction of the subcontig:
  1) Successfully adding forward/backward extension to subcontig,
  2) Mark k-mer as USED

- Found USED k-mers in both directions of the sub-contig AND both of the neighboring sub-contigs are ACTIVE:
  1) Set own state to ABORTED,
  2) Pick another random seed from hash table
Communication-Avoiding de Bruijn graph traversal

Oracle

P₀

P₁

P₂

GAT  ATC  TCT  CTG  TGA

AAC  ACC  CCG

AAT  ATG  TGC
Communication-Avoiding de Bruijn graph traversal

- Insights for oracle graph partitioning:
  - The nucleodite diversity is similar for given organisms.
  - An oracle partitioning derived from one genome will work for others of the same species.

- Given a set of contigs, generate a function $f_{oracle}$ that maps k-mers to their contig ID (encode this function in a compact vector). Use $f_{oracle}$ to “tweak” the hash function in order to preserve locality.

- Two practical use case scenarios:
  - Exploring assemblies of multiple genomes of the same species.
  - Optimizing an assembly by iterating over multiple lengths for the k-mers.
Complete assembly of human genome in 8.4 minutes using 15K cores.

350x speedup over original Meraculous (took 2,880 minutes and a large shared memory machine).
• Complete assembly of wheat genome in **39 minutes using 15K cores**.
• Original Meraculous would require (projected time) **a week (~300x slower)** and a shared memory machine with 1TB memory.
Contig generation for metagenome on Cray XC30

<table>
<thead>
<tr>
<th>Cores</th>
<th>k-mer analysis</th>
<th>contig generation</th>
<th>file I/O</th>
</tr>
</thead>
<tbody>
<tr>
<td>10K</td>
<td>776.04</td>
<td>47.83</td>
<td>92.81</td>
</tr>
<tr>
<td>20K</td>
<td>525.34</td>
<td>31.02</td>
<td>95.42</td>
</tr>
</tbody>
</table>

- Generated contigs for this huge metagenome dataset in \(~11~\text{minutes}\) using 20K cores of Edison.

- The contig generation would require a shared memory machine with \(~15~\text{TB of memory}!\)
Speedup via Communication-Avoiding Traversal

<table>
<thead>
<tr>
<th>Cores</th>
<th>Graph traversal time (sec)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>no-Oracle</td>
<td>oracle-1</td>
</tr>
<tr>
<td></td>
<td>oracle-1</td>
<td>oracle-4</td>
</tr>
<tr>
<td>480</td>
<td>145.8</td>
<td>105.8</td>
</tr>
<tr>
<td>1,920</td>
<td>46.3</td>
<td>35.9</td>
</tr>
</tbody>
</table>

Up to 2.8x faster traversal!

<table>
<thead>
<tr>
<th>Cores</th>
<th>Off-node communication (% of total)</th>
<th>% reduction in off-node comm</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>no-Oracle</td>
<td>oracle-1</td>
</tr>
<tr>
<td></td>
<td>no-Oracle</td>
<td>oracle-1</td>
</tr>
<tr>
<td>480</td>
<td>92.8 %</td>
<td>54.6 %</td>
</tr>
<tr>
<td>1,920</td>
<td>97.2 %</td>
<td>54.5 %</td>
</tr>
</tbody>
</table>

Up to 75.5% reduction of off-node communication!
Summary

• Presented core parallel algorithms in HipMer: a **High-performance** implementation of the **Meraculous** genome assembler.

• These algorithms scale to tens of thousands of cores and yield performance improvements from **days/weeks** down to **minutes**.

• HipMer makes the impossible possible
  • Breaks the limitation imposed by special hardware with huge memory
  • 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/](https://sourceforge.net/projects/hipmer/)

• Ongoing work: Adapt HipMer for metagenomic analysis.
Questions

Thank you!