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

Aydın Buluç
Computational Research Division, LBNL

May 23, 2016
Invited Talk at HiCOMB 2016
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
1. Three copies of the same novel.

For all men tragically great are made so through a certain morbidness... all mortal greatness is but disease.

2. Some text from the novel. All pages will be randomly cut into strips of characters. Random **typos (errors)** throughout each novel.

3. A few strips of characters from one page.

For all men tragically great are made so through a certain morbidness... all mortal greatness is but disease.

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 through a certain morbidness... all mortal greatness is but disease.

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.

3. The sequence is read into smaller pieces (**reads**). Can not read whole DNA sequence in one go.

4. All reads

5. Reconstruct original DNA sequence from the read set.
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

1. Chop reads into k-mers, process k-mers to **exclude errors**
   - I/O, bandwidth, and memory intensive

2. Construct & traverse de Bruijn graph of k-mers, generate contigs
   - **Latency bound (irregular accesses)**

3. Leverage read information to link contigs and generate scaffolds.
   - **Compute and I/O intensive**
Meraculous parallelization with UPC

- UPC is a PGAS (Partitioned Global Address Space) 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 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

Hyperloglog

G = Global cardinality estimation

Bloom Filter of size (G/n)

Also identify high-frequency k-mers at this step
Parallel k-mer analysis: pass 2

Reads

<table>
<thead>
<tr>
<th>Parse to k-mers</th>
<th>Hash k-mers &amp; find owners</th>
<th>All-to-all communication of k-mers</th>
<th>Store a k-mer in the local set only if it was seen before</th>
<th>The local set does the actual counting of the k-mers</th>
</tr>
</thead>
<tbody>
<tr>
<td>P₀</td>
<td>Hash k-mers</td>
<td>Received k-mers</td>
<td>Bloom Filter</td>
<td>Local set</td>
</tr>
<tr>
<td>P₁</td>
<td>Hash k-mers</td>
<td>Received k-mers</td>
<td>Bloom Filter</td>
<td>Local set</td>
</tr>
<tr>
<td>...</td>
<td>Hash k-mers</td>
<td>Received k-mers</td>
<td>Bloom Filter</td>
<td>Local set</td>
</tr>
<tr>
<td>Pₙ</td>
<td>Hash k-mers</td>
<td>Received k-mers</td>
<td>Bloom Filter</td>
<td>Local set</td>
</tr>
</tbody>
</table>

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
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

Local set

Keep track of the number of occurrences of each extension for each k-mer

Local set

Keep track of the number of occurrences of each extension for each k-mer

Local set

Use a threshold & find the high quality extensions of k-mers

Local set

Keep track of the number of occurrences of each extension for each k-mer

Local set

Keep track of the number of occurrences of each extension for each k-mer

Local set

Keep track of the number of occurrences of each extension for each k-mer

Local set
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.
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.
• **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 $\rightarrow 10x-20x$ performance improvement.
Aggregating stores optimization

\(P_0\) stores the k-mers & extensions in its local buckets in a lock-free & communication-free fashion.
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
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:

$\text{CGTATTGCCAATGCAACGTATCGGCCCAATCCGAT}$

$P_i$ 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
To be reused for metagenomics w/ modifications
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 $k$

<table>
<thead>
<tr>
<th>Read ID</th>
<th>start-pos</th>
<th>end-pos</th>
<th>Contig ID</th>
<th>start-pos</th>
<th>end-pos</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read 42</td>
<td>1</td>
<td>4</td>
<td>Contig 101</td>
<td>152</td>
<td>155</td>
</tr>
<tr>
<td>Read 42</td>
<td>130</td>
<td>150</td>
<td>Contig 61</td>
<td>1</td>
<td>21</td>
</tr>
<tr>
<td>Read 90</td>
<td>1</td>
<td>150</td>
<td>Contig 500</td>
<td>101</td>
<td>250</td>
</tr>
</tbody>
</table>
Building seed index

Seed Index

Target 0: ATG

Target 1: GCCA
Building seed index

Seed Index

Target 0: A C T G G

Target 1: G G C A
Building seed index

Seed Index

Seed: ACT

Seed: CTG

Target 0: ACTGG

Target 1: GGCAG
Building seed index

Seed Index

Seed: ACT

Seed: CTG

Seed: TGG

Target 0: A C T G G

Target 1: G G C A
Building seed index

Seed Index

Seed: ACT
Seed: GGC
Seed: CTG
Seed: TGG

Target 0: A C T G G

Target 1: G G C A
Building seed index

Seed Index

Seed: ACT
Seed: GGC
Seed: CTG
Seed: TGG
Seed: GCA

Target 0: ACTGG
Target 1: GGCA

Seed Index
Lookup seed index

Seed: ACT
Seed: GGC
Seed: CTG
Seed: TGG
Seed: GCA

Target 0:
ACTGG

Target 1:
GGCGA

query’s seed

query sequence: GCTG

Seed GCT is not found in seed index
Lookup seed index

- **Seed: ACT** → Target 0
- **Seed: GGC** → Target 0
- **Seed: CTG** → Target 0
- **Seed: TGG** → Target 1
- **Seed: GCA** → Target 1

**Query sequence**: G C T G

**Seeds**: ACT, GGC, CTG, TGG, GCA

**Targets**: 0, 1

**Query's seed**: G C T G

**Seed Index**

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

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

Data sizes
10TB
1TB
100GB

k-mer analysis → contig generation → alignment → gap closing

Increase k and iterate

Secret metagenome sauce #1
Iterative contig generation
Iterative contig generation

k-mer analysis
Iterative contig generation

k-mer analysis

De Bruijn graph traversal

Iterative contig generation
Iterative contig generation

1. k-mer analysis
2. De Bruijn graph traversal
3. Bubble merging and hair removal
4. Iterative contig generation

Iterate for larger k
Iterative contig generation

1. k-mer analysis
2. De Bruijn graph traversal
3. Bubble merging and hair removal
4. Iterative graph pruning
Iterative contig generation

1. k-mer analysis
2. De Bruijn graph traversal
3. Bubble merging and hair removal
4. Iterative graph pruning
5. Local Assembly
Iterative contig generation

- k-mer analysis
- De Bruijn graph traversal
- Bubble merging and hair removal
- Iterative graph pruning
- Local Assembly

Iterate for larger k
HipMer on a MOCK community

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

<table>
<thead>
<tr>
<th>Statistics</th>
<th>metaHipMer</th>
<th>metaSPAAdes</th>
</tr>
</thead>
<tbody>
<tr>
<td># contigs</td>
<td>2,670</td>
<td>5,100</td>
</tr>
<tr>
<td>Total length &gt; 10 kbp</td>
<td>92,245,198</td>
<td>92,152,728</td>
</tr>
<tr>
<td>Total length &gt; 50 kbp</td>
<td>69,493,125</td>
<td>77,292,121</td>
</tr>
<tr>
<td>Misassemblies</td>
<td>58</td>
<td>134</td>
</tr>
<tr>
<td>Mismatches per 100 kbp</td>
<td>3.48</td>
<td>77.05</td>
</tr>
<tr>
<td>Genomes fraction (%)</td>
<td>92.10</td>
<td>91.10</td>
</tr>
</tbody>
</table>
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).