De-novo Micro-assembly on the GPU

Lecture by: Victoria Popic
Scribed by: Laza Upatising
March 5, 2015

Review of DNA Sequencing

- Starting with the DNA molecule, we break them into fragments, then we sequence the fragments using modern sequencing technologies (such as Illumina), which reads approximately 150 base pairs at a time, with some known error rate.
- For sequencing a new organism, we use an algorithm called *De Novo Assembly*, which attempts to connect the reads to form the entire DNA sequence. When sequencing previously sequenced organism, we simply try to *align* reads with known templates of the organism's DNA.

De Novo Assembly (Novel organism)

Objective: Assemble a contiguous genome sequence from a set, R, of short reads. Note that the size of R may be extremely large, in the billions.

Method:

1. Construct the de Bruijin graph from the reads.
2. Reconstruct the DNA by finding the Euclidian path through the de Brujian graph.

De Brujin graph description:

- Node: All distinct k-mers in R. Theoretically, there are at most 4^K nodes.
- Edges: Each (k+1)-mer forms an edge between its length k prefix and length k suffix.

Alignment (Previously sequenced organism)

We can use many existing algorithms, such as BWT and hashing, to align reads. There alignments may contain inexact matches due to read errors and genomic variations. Genomic variations can be anything from Single Nucleotide Polymorphisms (SNPs), Deletions, Insertions, Inversions, Translocations, Duplications, etc…

When performing alignment, errors at the beginning or end of a read are problematic. In most alignment algorithms gaps are usually penalized much more heavily than mismatches. Insertion and deletions at the end of reads can result in misalignment. In the example given below, Figure 1 shows the true alignment, but most alignment algorithms will choose the result in Figure 2 since mismatches are preferred over gaps.

ACTGCGTCGTATATAT ACT - - - TCGTATATAT True Alignment Figure 1	GCGTCGTATATAT **ACT** TCGTATATAT Selected Alignment Figure 2

To reduce the number of misalignments, we should add a realign step to correct the mappings using the information about the alignment of other reads. A read may have inDels at the edges, which causes alignment problems, but we can use another overlapping read which have the same inDels in the middle. We will call this technique *Assembly INSPIRED Realignment*, introduced next.

Assembly INSPIRED Realignment

- Pick a window of a known genome.
- Discard every read that doesn't fall in that region.
- Assemble the region from scratch, as if assembling the genome of a novel algorithm.

To assemble a region of the genome from scratch, we'll first build a de Brujin graph of the genomic region in question, and then find the likelihood of paths through the de Brujin graph. To assembly a genomic region from scratch, we'll use the *De Novo Micro-assembly* algorithm. Note that reassembly is a part of a larger process, the Broad Institute Pipeline, GATK. The Variant calling workflow involves a step called HaplotypeCaller, which calls SNPs and inDels using our local reassembly technique to be discussed below.

De novo Micro-assembly algorithm

Objective: Reassemble a portion of the genomic sequence of a known organism based on high coverage erroneous reads.

1. Prepare the sequences for assembly. Gather the reads and the reference genome. We'll call the reads and reference genome collectively as sequences.
2. Build the de Brujin graph, G
    a. Find repeat kmers.
    b. For each sequence S, extend the de Brujin graph G using information in S. More details below.
3. Check for cycles in G
4. Prune the graph and recover dangling-chains.
5. Find the best paths through the graph, finishing the reassembly.

Given a sequence S, and existing de Brujin graph G, how do we extend the graph such that it encapsulates the information in S. First, a few definitions:

Repeat kmer: Kmers that appear more than once in at least one sequence.

Unique kmer: Kmers that appear at most once in any sequence.

Note that repeat kmers are problematic in de Brujin graphs since they create cycles.

Constructing G, with S:

Note that we have two pieces of auxiliary information: the repeat kmers and a map that we'll build <key = unique kmer, value = node name>, we'll call this the unique kmer map.

1. Graph initialization: For each repeat kmer, form dangling nodes representing that repeat kmer in the graph.
2. For each sequence S in all the sequences to assemble:
    a. Start our processing at the first unique kmer in S.
    b. Denote the current kmer we are processing as K_i. The next kmer as K_{i+1}.
    c. Ensure that kmers K_i and Ki_{+1} are nodes in the graph and in the unique kmer map (if the kmer is a unique kmer). If a kmer, K, does not exist in the graph or map, add a new mapping from <K, [new node name]> and add [new node name] to the graph.
    d. Denote the node representing kmer K_i as N_i and node representing kmer K_{i+1} as N_{i+1}.
    e. If an edge exists between N_i and N_{i+1} increase the count associated with that edge. Otherwise form an edge between N_i and N_{i+1} with count = 1.
    f. Go back to step b. and repeat until end of sequence.

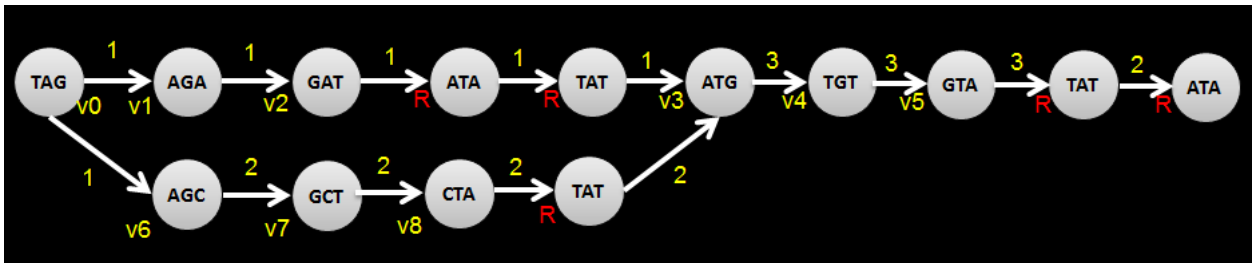
Due to the way the graph is constructed, there can be only one node for each unique kmer, but there can be more than one node representing a repeat kmer.

Example from slides:

Sequences to assemble:	Non-unique (repeat) kmers:
1. TAGATATGTATA 2. TAGCTATGTAT 3. AGCTATGTATA	TAT, ATA

Initially, the unique kmer map is empty.

After the insertion of all sequences in order, we have the following graph and unique kmer map:



As the number of sequences increases, the algorithm can take an increasingly long time. After a brief introduction to GPUs, we'll introduce a parallel algorithm to speed up the de Brujin graph construction.

A Brief Introduction to GPUs

GPUs were initially created to meet the demands of high performance computer graphics, specifically driven by the video game industry. Graphics computation requires a very high computation throughput since each pixel in a computer image much be calculated. This resulted in a divergence in high level architectural design decisions between the GPU and CPU. Whereas the CPU focuses on running one program extremely fast (very low computational latency), GPUs focuses on getting a lot of computation done in as little time as possible (very high computational bandwidth). The analogy given is the following scenario: You want to move a packages from location A to B. A CPU is like a ferrari, it can move a small number of packages from A to B extremely quickly, but only one or two packages at a time. A GPU is a fleet of motorcycles: each vehicle can only move one package much slower than the ferrari, but the entire motorcycle fleet can move more packages from A to B for a given period of time.

Today, the most widely used GPU programming paradigm is created by NVIDIA, called CUDA. It is based on the C and C++ programming languages and provides extensions to the language to allow GPU device programming. A function ran in parallel on the GPU is called a kernel. The device is divided up into a grid, which is a 3D array of thread blocks. Each thread block is a two dimensional array of threads. A thread is one execution context. Multiple threads in a thread block all run the same kernel - providing us with the parallel computing power.

Parallel de Brujin Graph Construction

Objective: An algorithm that can can correctly construct the de Brujin graph but is also highly parallelizable.

Notation: U = unique kmer, R = repeat kmer. U-U represents an edge between two unique kmers.

Output: CSR representation of the de Brujin graph.

CSR Graph Representation

Graph is represented by four arrays named nodes, offsets, uniqueID and edge_counts.

- The node array is simply a list of all nodes in the graph.
- For index i in the offset array:
  uniqueID[offset[i]],
  uniqueID[offset[i]+1], ⋯,
  uniqueID[offset[i+1]]
  represents edges:
  nodes[i] -> nodes[uniqueID[offset[i]]],
  nodes[i] -> nodes[uniqueID[offset[i]+1]], ⋯,

nodes[i] -> nodes[uniqueID[offset[i+1]]].
Where each edge has count:
edge_counts[uniqueID[offset[i]]],
edge_counts[uniqueID[offset[i]+1]], ⋯,
edge_counts[uniqueID[offset[i+1]]]

- Note that this means offsets[i+1] > offset[i] for all i.

Parallel de Brujin Graph Construction Algorithm Outline

1. Prepare sequences to be assembled.
2. Extract node kmers
3. Partition kmers into unique kmers and repeat kmers.
4. Extract unique kmers and add them to the nodes vector.
5. FInd the number of times each repeat kmer is duplicated and extract each instance into the nodes vector.
6. Extract edges involving repeat kmers, U-R, R-U, R-R as 2 vectors of from-to node IDs.
7. Generate (k+1)mers (edges).
8. Filter to keep unique prefix edges: U-U and U-R.
9. Compute the out-degrees and offsets of all the edge types.
10. Populate the adjacency map.

Since we didn't have much lecture time, we'll only focus on node generation.

Parallel de Brujin graph node construction

1. Concatenate all sequences into one superstring, which we'll also call the global array, delimiting the end of each sequence with a special character. e.g. ATATATATATA | GCGCGCGCGCGCG | ATCGGATCGCGAT | CGAGGAGAGAG
2. Each kmer can be represented by a 4-tuple coordinate (x, y, z, w) where
   x = sequence id (sequence number in global array)
   y = position in sequence x
   z = global position (index of kmer in the global array)
   w = not covered.
3. Each base is represented with by two bits, and then pack the kmer along with its coordinate into a 64-bit double word, call this 64-bit double word the Kmer key. Note that this means there is a limitation on the length of your kmer, and the length of the global array. The creation of Kmer keys can be done in parallel.
4. Sort all the Kmer keys.
5. Augment each Kmer key with the a count per sequence value of 1. Then reduce all the kmers by key and sequence id. At this point, for each sequence we will have created a kmer key for each kmer in the sequence, with repeat kmers having a count per sequence value higher than 1.
6. Now reduce all the kmer keys just by their key. This will give us a count of how many sequences have a kmer and how many times does the kmer occur in all sequences. Note that

if the number of times the kmer occurs in all sequences is greater than the number of sequences that have that kmer, we know that the kmer is a repeat kmer by the pigeonhole principal.

7. Now we know whether each kmer is a unique or repeat kmer, for each unique kmer assign unique IDs to it. We have now almost constructed the uniqueID array in the CSR graph representation, we are still missing the nodes for repeat kmers.
8. For each repeat Kmer, create a SUPER-mer based on the unique kmer occuring prior to the repeat kmer in the super array and the unique kmer occuring after the repeat kmer in the super array. Note that the unique kmer may not come immediately before or after the repeat kmer since we may have a series of repeat kmers in a sequence. The supercoordinate's kmer is representated by the concatenation of the base pairs found in: unique kmer prior - repeat kmer(s) - unique kmer after.
9. Collapse super-mers with the same sequence id and base pairs.
10. Once we have obtained the SUPER-kmers and their coordinates, lexicographically sort the Kmers. Then find the max predecessor prefix overlap in the number of kmers.
11. Given the SUPER-mers and their overlap, we can determine the number of repeat nodes to extract out of each supermer and the number of edges to extract.

Now we have constructed all the nodes in the CSR representation of the de Brujin graph in parallel. The next step is to form edges between each node. However, we did not have enough time to cover the material and the lecture ended here.

<u>Summary</u>

- DNA sequencing for novel organisms: De Novo Assembly
- DNA sequencing for known organism: Alignment
    - Alignment is problematic when reads have inDel mutations at the edges.
    - A solution to this problem is to reassemble the DNA in partial regions, utilizing information from overlapping reads to remove the problem of read edge inDels. Algorithm is called De Novo Micro-assembly
- In order to perform De Novo Micro-assembly, we must first build the de Brujin graph from the reads. This process can take a very long time when the number of reads is large.
- Utilize GPUs and their massive computational throughput to parallelize de Brujin graph construction.
- Covered a parallel de Brujin graph node construction algorithm.