FFTLab: Genome Resequencing Pipeline using Signal Processing for Alignment & GraphLab for Assembly

(Project Report for Big Data, Small Languages, Scalable Systems)

Ayushi Sinha[†]
asinha8@jhu.edu

Shuya Chu[†]
schu14@jhu.edu

Yuge Gong[†]
ygong8@jhu.edu

[†]Department of Computer Science
The Johns Hopkins University
Baltimore, Maryland 21218

***Abstract*—In this project report, we present a new approach to the problem of genome resequencing. We approach the problem of aligning short DNA sequences, or reads, to a complete genome reference as a signal processing problem, and use GraphLab, a big data processing framework, to merge and assemble these alignments into a complete genome sequence.**

I. Introduction

Genome resequencing is a very important problem in computational genomics. Several sequencing technologies have come up in the past decade or so. These technologies produce several DNA sequences, or reads, representing different parts of the sequenced genome. These reads must then be assembled in the right order to generate a resequenced genome.

Genome resequencing is not a trivial task. DNA, the chemical that comprises a genome, is represented by four letters, A, C, T, and G. Inevitably. there is a lot of repetition in a DNA sequence representation. This makes it hard to assemble reads into a sequence. One approach to resequencing is matching or aligning the reads obtained from an organism to a known reference, that is, a standard or average genome sequence for the species that the organism belongs to. Once the best position for each read is obtained, these reads can be stitched together into a sequence.

Several methods have been presented that implement this procedure in different ways. However, there are certain expectations that the procedures must meet, regardless of how they are implemented. First, the reads must be aligned with some measure of accuracy. Although, the DNA of organisms belonging to the same species are 99% alike, each individual has a unique DNA sequence. Therefore, the alignment process must allow for some amount of mismatch, while remaining within a certain threshold. Second, this process of aligning and assembling must be fast, since DNA sequences for larger organisms are longer and more complicated, and require the resequencing process to be able to scale well.

We present a method for alignment using signal processing, which allows for inexact alignment of reads to references. Our method provides a confidence score for each alignment, which can be adjusted according to the desired accuracy of the alignments. These alignments are then pushed through a big graph processing framework called GraphLab in order to assemble the alignements together into one sequence.

II. Motivation

A. String Matching

DNA sequences are represented using four letters, A, C, T, and G. Therefore, the alignment of reads to references can be thought of as a string matching problem. This problem in turn can be approached in multiple different ways.

1) Brute Force String Matching: The most simple way to approach this problem is to check if the characters in the read match the corresponding characters in the reference, and repeat this process for each position in the reference. This approach is extremely slow, with a comlexity of $O(N * M)$, where N is the size of reads and M is the size of the reference. Even for organisms like viruses, which have short genomes, there are about $50,000$ characters in a reference, and about $10,000$ reads with an average length of about 100 letters. With increasing sizes of genome sequences, not only does the reference become longer, but the number of reads also increases. Therefore, this method will very quickly become prohibitively slow.

2) Fast String Matching: A slightly better way to approach this problem is to use the information obtained from mismatches constructively. Once a mismatch is encountered, we know that the substring ending at the mismatch is not a matching substring, unless it contains the starting letter of the read (include image). Therefore, if a mismatch is encountered, we can start comparing at the position after the mismatch, unless we encounter the corner case described. Therefore, the size of the shift in the case of a mismatch is determined by

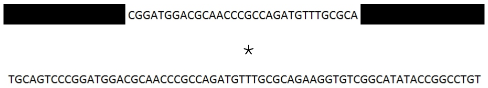

CGGATGGACGCAACCCGCCAGATGTTTGCGCA
*
TGCAGTCCCGGATGGACGCAACCCGCCAGATGTTTGCGCAGAAGGTGTCGGCATATACCGGCCTGT

Fig. 2: Cross correlation between read (top) and reference

Fig. 3: Cross-correlation response

the number of repetitions in the read, and is independent of the reference. In particular, this can be computed in $O(N)$ time. The per character comparison can still take $O(M)$ time in the worst case, that is, when there is a mismatch at every character in the reference, or at the last character of the read at every shift. Therefore, the time complexity of this approach is $O(N + M)$[1].

3) String Matching using Fast Fourier Transform: We can, however, do even better. Despite making use of information obtained from mismatches, in the worst case we can end up performing $M + N$ per character comparisons. However, this can be avoided by using signal processing to compare reads to references. We can compute the fourier transform of the ASCII representations of the reads and the reference, and perform a restricted or windowed pattern matching, where the read is compared with contiguous substrings of the reference of length N. This computation entails a cross correlation per comparison, which is simply multiplication in Fourier space. Therefore, alignment can be performed using this approach in $O(N \log M)$ time. This method will be described in detail in Section III.

B. Assembly

Once we have all the alignments, we want to merge these alignments together to form contigs, and in turn piece the contigs together to form scaffolds, and finally obtain a fully resequenced genome. Crossbow[2] performs this procedure using Hadoop MapReduce. We use GraphLab instead, not only because GraphLab claims to be faster than Hadoop MapReduce for many applications [4], but also because GraphLab is able to perform this process in a more neighborhood aware sense than Hadoop MapReduce. GraphLab checks relevant reads only, rather than all reads, for overlap. That is, it checks only reads with overlap for merging to form contigs, and then orders these contigs into scaffolds, until a fully resquenced genome is obtained.

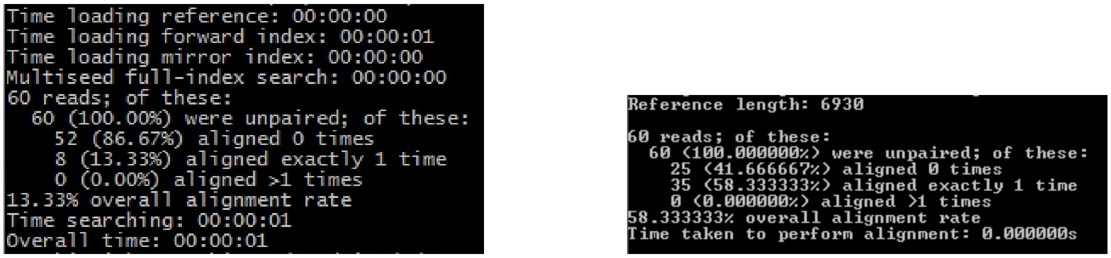
GraphLab supports dynamic and graph-parallel computation, and it also ensures data consistency and achieves a high degree of parallel performance in shared memory. We hope that this will help accelerate the assembly of large genome data. The design and implementation of graph-parallel algorithms is easily simplified by GraphLab. Therefore, it allows the user to focus on defining the operation on per vertex data. That is, the user focuses not on parallel computation, but on sequential computation at each vertex. Each vertex is computed in parallel by GraphLab at runtime, while at the same time supporting sequential shared memory abstraction. This allows each vertex to read and write data on adjacent vertices.

III. Method

A. String Matching using Fast Fourier Tranform

We can think of the alignment problem as a pattern matching problem, where the pattern is a 1-dimensional array containing some combination of the letters A, C, T, and G. The problem statement is: *Given an instance of a read, find all occurances of the read within a reference*.

The intuition here is to compute the cross-correlation of the read with the reference, and look for local maxima, since a match corresponds to high response in a cross-correlation. Given the fourier transform of the read, $f[]$, and of the reference, $g[]$, the cross-correlation corresponds to computing the dot product of $f[]$ and $g[]$ (Fig. 2). However, simply computing this dot product results in high-response everywhere (Fig. 3). The cross-correlation has large values because the dot product of the reference with the translated read instance is large. This is because the dot product is large not only when the the values of $f[]$ are similar to the values of $g[]$, but also when the values of $g[]$ are large. Since we are looking at the ASCII representation of the characters representing the read and the reference, the values of $g[]$ are always large, resulting in high response everywhere. Therefore, this process produces several alignments, many of which are not of good quality or accuracy.



```
Time loading reference: 00:00:00
Time loading forward index: 00:00:01
Time loading mirror index: 00:00:00
Multiseed full-index search: 00:00:00
60 reads; of these:
  60 (100.00%) were unpaired; of these:
    52 (86.67%) aligned 0 times
    8 (13.33%) aligned exactly 1 time
    0 (0.00%) aligned >1 times
13.33% overall alignment rate
Time searching: 00:00:01
Overall time: 00:00:01
```

(a) Bowtie2

```
Reference length: 6930

60 reads; of these:
  60 (100.000000%) were unpaired; of these:
    25 (41.666667%) aligned 0 times
    35 (58.333333%) aligned exactly 1 time
    0 (0.000000%) aligned >1 times
58.333333% overall alignment rate
Time taken to perform alignment: 0.000000s
```

(b) Cross-correlation

Fig. 1: Alignment results for reference of length 6930 and 60 reads

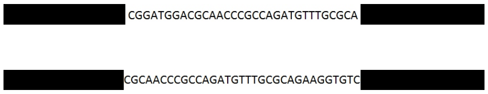

CGGATGGACGCAACCCGCCAGATGTTTGCGCA
CGCAACCCGCCAGATGTTTGCGCAGAAGGTGTC

Fig. 5: Windowed cross correlation between read (top) and reference

Fig. 6: Windowed cross-correlation response showing one position of high response

A comparison with Bowtie2[3] is shown in Fig. 1. Aligning 60 reads of a sequence of the lambda virus genome to a reference of length 6930 results in less the 15% alignments using bowtie, whereas the cross-correlation results in over 50% alignments. We do not want to measure how correlated the read instance is to a region in the reference. What we do want to measure is how similar the read instance is to a region in the reference. In other words, for every point p in the reference, we want to know how similar the region about p is to the translated read. This can be formally expressed as follows: for every p, we would like to compute

$$||\rho_p(f[]) - \rho_p(\chi[]).g[]||^2. \tag{1}$$

Here, ρ_α is the unitary representation that shifts an array by α indices, that is,

$$\rho_\alpha(f[])[k] = f[k-\alpha].$$

We know that for two arrays $f[]$ and $g[]$, we can define $f[].g[]$ to be the entry-wise product of the two arrays. That is,

$$\begin{aligned}(f[].g[])[k] &= f[k].g[k]\\ \Rightarrow \rho_\alpha(f[].g[])[k] &= f[k-\alpha].g[k-\alpha]\\ &= \rho_\alpha(f[])[k].\rho_\alpha(g[])[k]\\ \Rightarrow \rho_\alpha(f[].g[]) &= \rho_\alpha(f[]).\rho_\alpha(g[])\end{aligned}$$

Therefore, in Equation (1), $\rho_p(f[])$ represents the translation of the read $f[]$ to be centered around the point p. $\chi[]$, on the other hand, is the characteristic grid of the read

$$\chi[k] = \begin{cases} 1, & \text{if } k \text{ is inside the read}\\ 0, & \text{otherwise}\end{cases}$$

$\rho_p(\chi[])$ shifts this characteristic grid to be centered around the point p. The characteristic grid is also of the same length as the reference, and hence, multiplying $\rho_p(\chi[])$ with $g[]$ zeros out all of $g[]$ except for the region about p. Therefore, the term $\rho_p(\chi[]).g[]$ in Equation (1) represents the restriction of $g[]$ to the region about the point p. Further, Equation (1) can be simplified for computation:

$$\begin{aligned}||\rho_p(f[]) - \rho_p(\chi[]).g[]||^2 =&< \rho_p(f[]), \rho_p(f[]) >\\ &- 2 < \rho_p(f[]), \rho_p(\chi[]).g[] >\\ &+ < \rho_p(\chi[]).g[], \rho_p(\chi[]).g[] >\end{aligned} \tag{2}$$

Here, using the fact that the representation ρ_p is unitary, we can say

$$< \rho_p(f[]), \rho_p(f[]) >= ||f[]||^2. \tag{3}$$

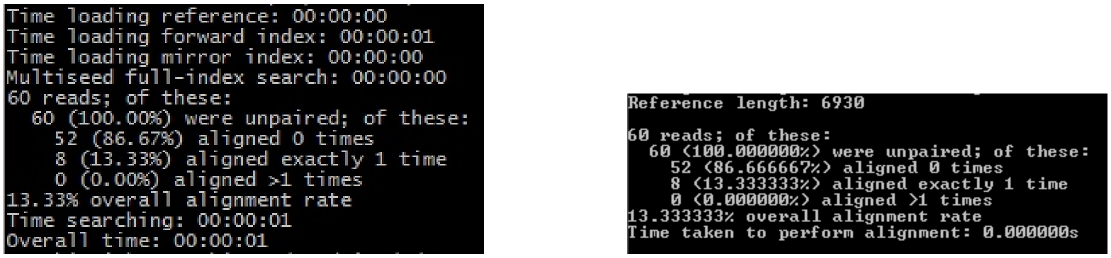
Further, using the fact that $\chi[]$ is real-valued, we can move it to the other side of the dot-product.

$$\begin{aligned}< \rho_p(f[]), \rho_p(\chi[]).g[] > &=< \rho_p(\chi[]).g[], \rho_p(f[]) >\\ &= \sum_{k=0}^{n-1} (\rho_p(\chi[]).g[])[k].\rho_p(\bar{f}[])[k]\\ &= \sum_{k=0}^{n-1} (\rho_p(\chi[])[k].g[k].\rho_p(\bar{f}[])[k]\\ &= \sum_{k=0}^{n-1} g[k].(\rho_p(\bar{\chi}[])[k].\rho_p(\bar{f}[])[k]\\ &= \sum_{k=0}^{n-1} g[k].(\rho_p(\chi[])\bar{.\rho}_p(f[])[k]\\ &=< g[], \rho_p(\chi[]).\rho_p(f[]) >\\ &=< \rho_p(\chi[].f[]), g[] >\end{aligned} \tag{4}$$

Since $\chi[]$ is equal to one whenever $f[]$ is non-zero, we can simply say

$$< \rho_p(f[]), \rho_p(\chi[]).g[] >=< \rho_p(f[]), g[] > . \tag{5}$$

In the same way as Equation (4), we can use the fact that both $\rho[]$ and $g[]$ in the term $< \rho_p(\chi[]).g[], \rho_p(\chi[]).g[] >$ are real-valued, and move them to the other side of the dot product to



```
Time loading reference: 00:00:00
Time loading forward index: 00:00:01
Time loading mirror index: 00:00:00
Multiseed full-index search: 00:00:00
60 reads; of these:
  60 (100.00%) were unpaired; of these:
    52 (86.67%) aligned 0 times
    8 (13.33%) aligned exactly 1 time
    0 (0.00%) aligned >1 times
13.33% overall alignment rate
Time searching: 00:00:01
Overall time: 00:00:01
```

(a) Bowtie2

```
Reference length: 6930

60 reads; of these:
  60 (100.000000%) were unpaired; of these:
    52 (86.666667%) aligned 0 times
    8 (13.333333%) aligned exactly 1 time
    0 (0.000000%) aligned >1 times
13.333333% overall alignment rate
Time taken to perform alignment: 0.000000s
```

(b) Windowed cross-correlation

Fig. 4: Alignment results for reference of length 6930 and 60 reads

get

$$< \rho_p(\chi[]).g[], \rho_p(\chi[]).g[] >=< (\rho_p(\chi[]))^2, g^2[] > .$$

Finally, since $\chi[]$ is always equal to either 0 or 1, we know that $\chi^2[] = \chi[]$.

$$< \rho_p(\chi[]).g[], \rho_p(\chi[]).g[] >=< \rho_p(\chi[]), g^2[] > . \qquad (6)$$

Combining Equations (3), (5) and (6), we can rewrite Equation (2) as

$$\begin{aligned} ||\rho_p(f[]) - \rho_p(\chi[]).g[]||^2 = ||f[]||^2 + &< \rho_p(\chi[]), g^2[] > \\ &- 2 < \rho_p(f[]), g[] > \\ = ||f[]||^2 + \chi[] * g^2[] - 2f[] * g[], \end{aligned}$$

where $\chi[] * g^2[]$ is the windowed norm, and $2f[] * g[]$ is the moving dot product.

Computing this windowed moving dot product (Fig. 5) results in high response only at the exact match or best match location, while everything else produces low response, making it easy to single out the best match, if there is one (Fig. 6). This method makes finding inexact best matches extremely efficient. It does not involve any backtracking in depth first search tree[3]. Simply lowering the response threshold for best match will include inexact matches into the alignment, with the threshold acting as the tolerance for mismatches. We use this response for each alignment as the quality measure for the alignment. Fig. 4 shows a comparison between Bowtie2 and our method. We ran both methods on a reference of length 6930 and 60 reads, and both methods gave us the exact same result of 8 alignments.

B. Assembly using GraphLab

Once we have all the aligned reads, there is a need to merge and assemble these reads into a genome sequence efficiently. In order to do this, we use the GraphLab API, which exposes an MPI-like single program multiple data (SPMD) interface [5]. That is, GraphLab simulates that state where each node in a distributed cluster appears to run the same operation in lock-step. But this is not always the case. Different operations might have different complexity, and may even run asynchronously. Therefore, to support the simulation GraphLab is trying to achieve, we need to implement certain functions for GraphLab to implement. For our particular application, we implement a function to find the best path in a graph of overlapping reads, and another to assemble the best path into a genome sequence.

Our input for GraphLab is a text file containing a list of aligned reads obtained from the alignment procedure explained above. These reads are sorted by the starting position of the alignment, or offset, with respect to the reference. The input file contains not just the aligned reads, but also, for each read, a read ID, its length and quality of alignment, and the read IDs of any following reads it overlaps with. From this input, we want a sequenced genome that maximizes the the quality of alignment scores.

We define our distributed data graph $G = (V, E)$ such that each vertex V represents an aligned read, and each edge E joins consecutive vertices, and also those vertices that represent reads with overlaps. G is, however, a directed graph. So there is only an edge pointing in the forward direction, that is, there is an edge between two vertices if and only if a read overlaps with any following read. An edge between every consecutive vertex is required because GraphLab terminates at a vertex with no outgoing edges. Each vertex is uniquely identified by the read ID, and also contains the read, its length, the offset, and its quality of alignment.

Several reads overlap with several other reads, and we have to merge the best aligned reads together to form our contigs. There multiple overlaps imply that there can be several outgoing edges from many vertices. Our aim is to use the quality score from the alignment process to choose the best path from the source vertex, or the first aligned read, to the destination vertex, or the last aligned read. The best path should maximize the quality score. Each vertex chooses its best neighbor independently from other vertices. Therefore, we can merge the content of each vertex pair independently and recursively to converge to the best sequenced genome.

For instance, if we have a graph as in Fig. 7, we merge the contents of vertex R_1 and R_2 if R_1 and R_2 overlap, and store the merged contents in R_1. Similarly, as long as there is overlap, the contents of R_2 and R_3 are merged into R_2, and the contents of R_3 and R_4 are merged into R_3. These operations are called *vertex programs*, or short programs or operations that are executed on a vertex in G [5]. Each vertex program performs the following three phases of execution [5]:

1) Gather Phase: This phase calls the *gather()* function, which returns all the vertices which need to update their data;

2) Apply Phase: This phase calls the *apply()* function, which applies the changes and updates the vertices;

3) Scatter Phase: This phase calls a *scatter()* function, which returns all vertices that need to be updated due to the changes in the apply phase.

If there is no overlap, the two vertices are connected with 'N's. After this step, the new contents of R_1 and R_2 are merged into R_1 and similarly with R_2 and R_3. However, the content of R_4 did not change. We keep track of unchanged vertices using dynamic scheduling, and do not remerge the contents of unchanged vertices. In this way, we recursively merge contents of vertices until the contents in a vertex remain unchanged. That is, we stop when the scatter phase returns no vertex that needs to be updated. At this point, we have the fully merged content in vertex R_1. In other words, our graph will converge from bottom to top.

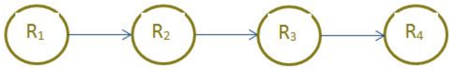


Fig. 7: An example graph in GraphLab

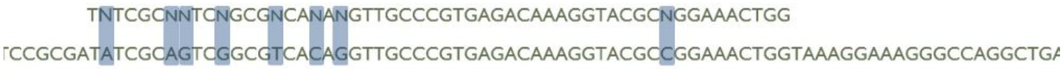

TNTCGCNNTCNGCGNCANANGTTGCCCGTGAGACAAAGGTACGCNGGAAACTGG
TCCGCGATATCGCAGTCGGCGTCACAGGTTGCCCGTGAGACAAAGGTACGCCGGAAACTGGTAAAGGAAAGGGCCAGGCTGA

Fig. 8: An alignment using our method: read (top) and reference (below)

IV. Results

Our alignment process gave us results that matched the state-of-the-art. For all of our experiments, we used the range $(0.918, 1.082)$ as the response range for accepted alignments, and compared our results against Bowtie2. We disabled reverse alignment in Bowtie2, and also increased the length of each seed to the maximum allowed seed length, in order to minimize multiple seed alignment, because we have not yet implemented these features using our method. In each experiment, we obtained the same number of alignments as Bowtie2. However, we have no yet implemented multiple seed alignment, which Bowtie does implement. This implies that we find more exact matches that Bowtie2. For instance, Bowtie2 is not able to align the alignment shown in Fig. 8, but our method can. A closer look shows that each of the mismatches in the inexact alignment is a mismatch with 'N'. Once we have implemented multiple seed alignment, we should have more total alignments than Bowtie2.

The disadvantage of our method is that it is slower than Bowtie2. As shown in Fig. 9, it takes our method about two and a half minutes to align reads from a lambda virus to its reference, whereas it takes Bowtie2 one second. However, the runtime of Bowtie2 deteriorates quickly with edit distance, or increased leniency in mismatches. That is, in order for Bowtie2 to find the alignment in Fig. 8, the edit distance would have to be increased, and the runtime would also suffer. Whereas, the runtime of our method does not depend on the edit distance. In order to increase the leniency in mismatches, we simply have to increase the response range for accepted alignments. Our method is also not optimized yet, and we will discuss ideas to improve runtime in the future works section.

The assembly algorithm successfully produces fully sequenced genome data, but also suffers from very slow runtime. For a small lambda virus genome data, it takes approximately 4 hours to assemble the alignments. Again, we haven't optimized our code, and the cluster we ran our code on could not see MPI, although MPI was installed on the cluster. This could be the reason why our assembly code is slow. Running the recursive framework of GraphLab sequentially, rather than in parallel, would run painfully slow. We also have not considered other parallel processing frameworks for this procedure, in case GraphLab is not the most efficient framework for this kind of application.

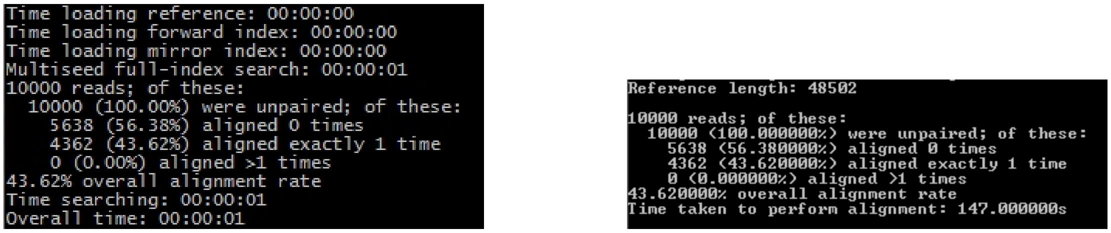
V. Limitations

One of the limitations of our work currently is that GraphLab cannot handle negative offset. So, if the first read, in terms of starting position, aligns with the reference in such a way that the starting position of the read with respect to the reference is negative (for instance, if the bottom read in Fig. 10 was the start of the reference, and the top read was aligned as shown in Fig. 10), then we currently manually trim the first aligned read and change the offset to 0.

Another limitation is that you have to manually enter the first and last read IDs in the input file to run the assembly code on GraphLab. This can be fixed if the alignment code output the first and last aligned reads, sorted with respect to their alignment positions with the reference, and if the assembly code could read this output in.

VI. Future Work

When we compute the similarty between $f[]$ and $g[]$ using signal processing, we are in a way computing how close the values of $f[]$ and $g[]$ are. Since we are using the ASCII presentation of the reads and reference in our computation, our method will penalize a mismatch between A and G or C and T more than it will penalize a mismatch between A and C. However, most errors in genome sequencing are the miscoding between C and T, and G and A [6]. That is, a mismatch between A and G, and between C and T should be penalized less than a mismatch between A and C. Therefore, coming up with a better representation should give us better inexact matches.



```
Time loading reference: 00:00:00
Time loading forward index: 00:00:00
Time loading mirror index: 00:00:00
Multiseed full-index search: 00:00:01
10000 reads; of these:
  10000 (100.00%) were unpaired; of these:
    5638 (56.38%) aligned 0 times
    4362 (43.62%) aligned exactly 1 time
    0 (0.00%) aligned >1 times
43.62% overall alignment rate
Time searching: 00:00:01
Overall time: 00:00:01
```

(a) Bowtie2

```
Reference length: 48502

10000 reads; of these:
  10000 (100.000000%) were unpaired; of these:
    5638 (56.380000%) aligned 0 times
    4362 (43.620000%) aligned exactly 1 time
    0 (0.000000%) aligned >1 times
43.620000% overall alignment rate
Time taken to perform alignment: 147.000000s
```

(b) Windowed cross-correlation

Fig. 9: Alignment results for reference of length 48502 and 10000 reads

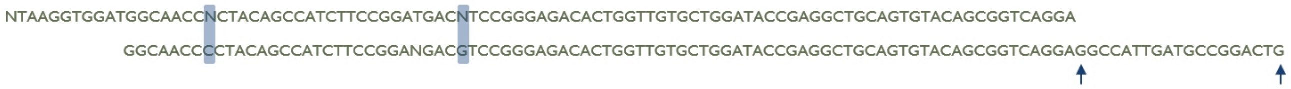

NTAAGGTGGATGGCAACCNCTACAGCCATCTTCCGGATGACNTCCGGGAGACACTGGTTGTGCTGGATACCGAGGCTGCAGTGTACAGCGGTCAGGA
GGCAACCCCTACAGCCATCTTCCGGANGACGTCCGGGAGACACTGGTTGTGCTGGATACCGAGGCTGCAGTGTACAGCGGTCAGGAGGCCATTGATGCCGGACTG

Fig. 10: Two overlapping aligned reads

In the same vein, a mismatch between G and N or T and N will be penalized less than a mismatch between A and N or C and N. However, this is not always correct. For instance, in the case that the phred-scale base error probability is high for A or C, a mismatch between A and N or C and N should be penalized less because the confidence in A and C being correct is less than the confidence in G and T. However, we do not use this error probability for each character in the reads at all currently. We plan on using this information to compute the representations of A, C, G, T, and N, such that A and G and closer to each other, C and T are closer to each other, and the two characters with the highest error probability will be placed closer to N.

Further, one big reason for our alignment's slow speed is that our method performs a sequential alignment as the reads are streamed in to memory. A different option would be to read in all the reads and store them in a data structure, use the error probability to compute a costum representation for the reads and the reference based on the mean error probability for each character, and then run the alignment procedure on the reads in parallel.

We also hope to add more functionality to our alignment procedure, for instance, multiple seed alignment, reverse alignment, and several more. Once we add these features, we can perform a fair comparison against Bowtie2.

We also plan to try and optimize our assembly algorithm on GraphLab, or try to use a different big data processing framework that is also parallel, and see if it works better for our application. Our current implementation of the assembly procedure using GraphLab also throws out valueable information. Take, for example, the case in Fig. 10. Here, when the top read, r1, is merged with the bottom read, r2, we simply append the substring of r2 starting where r1 ends to the end of r2. Hence, we keep the 'N's if they were present in the top read, rather than replacing it with information from the bottom read. We plan to incorporate a way to find 'N's and replace them if we can.

In addition, we are currently using the stringstream class to read and write data into GraphLab. However, the GraphLab API suggest that stringstream is slow, and significant performance gains can be achieved by using faster ways to parse a string, for instance, through the use of C parsing or by using boost::spirit [5]. This additional optimization can make the code faster.

VII. Conclusion

We have described in this report a new pipeline to resequence genome data by aligning reads to references. We present a read aligner which uses fast fourier transforms to compute a windowed cross-correlation between reads and the reference to align reads to the reference. We then used the correlation as the quality score or accepted alignments, and used GraphLab to merge and assemble the reads into a fully sequenced genome. This pipeline exposes new ways to improve the alignment process, and new ways to think about the assembling these alignments into a resequenced genome.

Acknowledgment

The authors would like to thank Dr. Yanif Ahmad for his help with the project, Dr. Ben Langmead for helping us understand Bowtie2, and Dr. Misha Kazhdan for his slides on Fast Fourier Transforms.

References

[1] Knuth, D.E., Morris, J.H., Pratt, V.R. (1977). Fast Pattern Matching in Strings. *SIAM Journal on Computing*, 6: 323-350.

[2] Langmead, B., Schatz, M.C., Lin, J., Pop, M., Salzberg, S.L. (2009). Searching for SNPs with cloud computing. *Genome Biology*, 10:R134.

[3] Langmead, B., Salzberg, S.L. (2012). Fast gapped-read alignment with Bowtie 2. *Nature Methods*, 9:357-359.

[4] Low, Y., Bickson. D, Gonzalez, J, Guestrin, C, Kyrola, A, Hellerstein, J.M. (2012). Distributed GraphLab: a framework for machine learning and data mining in the cloud. *Proc. VLDB Endow*, 5, 8: 716-727.

[5] Basic GraphLab Tutorial. *GraphLab: Distributed Graph-Parallel API 2.2.*, Web. 16 Dec. 2013. http://docs.graphlab.org/index.html

[6] Gilbert, M. T. P., Binladen, J, Miller, W, Wiuf, C, Willerslev, E, Poinar, H, Carlson, J.E., Leebens-Mack, J.H., Schuster, S.C. (2006). Recharacterization of Ancient DNA Miscoding Lesions: Insights in the Era of Sequencing-by-synthesis. *ucleic Acids Research*, 35, 1: 1-10