Concept 1

MAIN IS NOT TRUE

NGS NGSnext-generation-sequencing

DBG DBGde-Bruijn graph

OLC OLCoverlap-layout-consensus

1.1 Pacbio reads

Daligner finds alignments between long, noisy reads. Pacific Biosciences has commercially launched its first sequencer in 2011. It is able to output reads with an average of 1000 bases, which is significantly longer than NGS (NGS) reads [?]. In 2014, a new polymerase-chemistry combination was released, called P6-C4. This version can output average read lengths of 10000-15000 bases, and its longest reads can exceed 40000 bases [?]. While the drawback is that these reads have an error rate of 12-15%, this can be compensated by the distribution of these errors [?]. First, the set of reads is a nearly Poisson sampling of the sampled genome. This implies that there exists a coverage c for every target coverage k, such that every region of the genome is covered k times [?]. Secondly, the work of Churchill and Waterman [?] implies that the accuracy of the consensus sequence of k sequences is $O(\epsilon^k)$, which goes to 0 as k increases. This means that if the reads are long enough to handle repetitive regions, in principle a near perfect de novo assembly of the genome is possible, given enough coverage [?].

Important points for de novo DNA sequencing are: what level of coverage is needed for high quality assembly? And how to build an assembler that is able to deal with high error rates and long reads? Most previous assemblers work with NGS reads, which are much shorter and have much lower error rates. Some algorithms used in these assemblers, such as DBG (DBG) [?] would grow too large for high error rates and long reads. Since Daligner was build, new methods of using DBG with long reads have been developed, but they rely on a short read based DBG to correct errors in long reads [?][?].

1.2 Daligner

The first step in an OLC (OLC) assembler is usually finding overlaps between reads [?]. BLASR [?] was the only long read aligner at the time, and inpsired Daligner. It uses the same filtering concept, but with a cache-coherent threaded radix sort to find seeds, instead of a BWT index [?]. The most time-consuming step is extending the seed hit to find an alignment. To do this, Daligner uses a novel method which adaptively computes furthest reaching waves of the older O(nd) algorithm [?], combined with heuristic trimming and a datastructure that describes a sparse path from the seed hit to the furthest reaching point.

Daligner performs all-to-all comparison on two input databases $\mathcal{A}$, with M long reads $A_1, A_2, ...A_M$ and $\mathcal{B}$, with N long reads $B_1, B_2, ...B_N$ over alphabet $\Sigma = 4$ It reports alignments $P = (a, i, g)x(b, j, h)$ such that $len(P) = ((g - i) + (h - j))/2 \geq \tau$ and the optimal alignment between $A_a[i + 1, g]$ and $B_b[j + 1, h]$ has no more than $2\epsilon \cdot len(P)$ differences, where a difference can be either an insertion, a deletion or a substitution. Both τ and ϵ are user settable parameters, where τ is the minimum alignment length and ϵ the average error rate. The correlation, or percent identity of the alignment is defined as $1 - 2\epsilon$.

An edit graph for read $A = a_1a_2...a_m$ and $B = b_1b_2...b_n$ is a graph with $(m+1)(n+1)$ vertices $(i, j) \in [0, M] \times [0, N]$. It also has three types of edges:

- deletion edges $(i - 1, j) \to (i, j)$ with label $\begin{bmatrix} a_i \\ - \end{bmatrix}$ if $i > 0$.
- insertion edges $(i, j - 1) \to (i, j)$ with label $\begin{bmatrix} - \\ b_j \end{bmatrix}$ if $j > 0$.
- diagonal edges $(i - 1, j - 1) \to (i, j)$ with label $\begin{bmatrix} a_i \\ b_j \end{bmatrix}$ if $i, j > 0$.

An alignment between $A[i + 1, g]$ and $B[j + 1, h]$ is described as a sequence of labels from vertex (i, j) to (g, h). A diagonal edge can be either be a match edge, when $a_i = b_j$, or a substitution edge. If a match edge has weight 0, and the other edges have weight 1, the weight of the total path is the number of differences in the alignment it represents. To find suitable alignments, we have to find a read subset pairs P such that $len(P) \geq \tau$ and the weight of the lowest scoring path between (i, j) and (g, h) in the edit graph of A_a and B_b is not more than $2\epsilon \cdot len(P)$.

The O(ND) algorithm tries to find progressive waves of furthest reaching (f.r.) points until the endpoint is reached. The goal is to find longest possible paths starting at a starting point $\rho = (i, j)$ with 0 differences, then 1 difference, then 2 and so on. After d differences, the possible paths can end in diagonals $\kappa \pm d$, where $\kappa = i - j$ is the diagonal of the starting point. The furthest reaching point on diagonal k that can be reached from ρ with d differences is called $F_\rho(d, k)$. A collection of these points for a particular value of d is called the d-wave emanating from ρ, and defined as $W_\rho(d) = \{F_\rho(d, \kappa - d), ..., F_\rho(d, \kappa + d)\}$. $F_\rho(d, k)$ will be refered to as $F(d, k)$, where ρ is implicitely understood from the context.

In the O(ND) paper it is proven that:

$$F(d, k) = Slide(k, max\{F(d-1, k-1) + (1, 0), F(d-1, k) + (1, 1), F(d-1, k+1) + (0, 1)\} \tag{1.1}$$

where $Slide(k, (i, j)) = (i, j) + max\{\Delta : a_{i+1}a_{i+2}...a_{i+\Delta} = b_{j+1}b_{j+2}...b_{j+\Delta}\}$.

A slide is a path of sequential match edges. The f.r. d-point on diagonal k is calculated by finding the furthest of

- the f.r. (d-1)-point on $k - 1$ followed by an insertion

- the f.r. (d-1)-point on k followed by a substitution
- the f.r. (d-1)-point on $k+1$ followed by a deletion

and then continuing as far as possible along the slide. A point (i, j) is furthest when its anti-diagonal $i + j$ is greatest. The best alignment between reads A and B is the smallest d such that $(m, n) \in W_{(0,0)}(d)$, where m and n are the length of reads A and B. The O(ND) algorithm cimputes d-waves from starting point $(0, 0)$ until the end point (m, n) is reached. The complexity of this algorithm is $O(n + d^2)$ when A and B are non-repetitive sequences [**?**]. Because seeds are not always at the beginning, so waves are computed in both forward and reverse direction. The latter is easily done by reversing the direction of edges in the edit graph.

1.3 Seeding: concept

To find suitable starting points for the edit graphs, seeding is done. A seed is a section where $A[i, g]$ and $B[j, h]$ have a certain high similarity that indicates that these reads probably originate from the same part of the genome. Finding a seed includes finding matching k-mers for every readpair (a, b) with $a \in \mathcal{A}$ and $b \in \mathcal{B}$. Previous methods to match k-mers include Suffix Arrays [**?**] and BWT indices [**?**].

Assuming that the k-mer matches are independent, the probablity that a k-mer is conserved while sequencing is $\pi = (1 - 2\epsilon)^k$. The number of conserved k-mers in an alignment of τ basepairs is a Bernouilli distribution with rate π, so an average of $\tau \cdot \pi$ k-mers are expected in this alignment. An example: $k = 14$, $\epsilon = 15\%$ and $\tau = 1500$, then $\pi = .7^{14} = 0.0067$ and the average number of conserved k-mers is 10. Only .046% of the expected readpairs have 1 or fewer k-mers, and only 0.26% have 2 of fewer. To filter with 99.74% sensitivity, only readpairs with 3 or more k-mer matches need to be examined.

The specificity of the filter is increased in two ways:

- computing the number of k-mer matches in diagonal bands of width 2^s instead of in the whole reads
- thresholding on the number of bases in k-mer matches, instead of the number of k-mers themselves

The first way decreases the false positive rate because it only allows readpairs that have their k-mer matches relatively close, indicating a smaller region with higher similarity. The second way relies on the fact that 3 overlapping k-mers have a higher probability ($\pi^{1+2/k}$) than 3 disjoint k-mers with $3k$ basepairs (π^3).

The actually find the k-mer matches, Daligner uses a sort-merge procedure:

- Build a list $List_X = \{(kmer(X_x, i), x, i)\}_{x,i}$ of all k-mers for database $X \in \{\mathcal{A}, \mathcal{B}\}$, where $kmer(R, i)$ is the k-mer $R[i - k + 1, i]$.
- Sort both lists in order of their k-mers.

- Merge the two lists and build $List_M = \{(a, b, i, j) : kmer(A_a, i) = kmer(B_b, j)\}$ of read and position pairs that have the same k-mer.
- Sort $List_M$ lexicographically on a, b and i where a is most significant.

All entries for a certain read pair (a, b) are in a continuous segment of the list. This makes it easy to determine if that read pair has enough k-mers and in the right places to constitute a seed hit. Given parameters h and s, each entry (a, b, i, j) for the current read pair is placed in diagonal bands $d = \lfloor (i-j)/2^s \rfloor$ and $d + 1$. Now determine the number of bases in the A-read covered by k-mers in each pair of neighbouring diagonal bands. Note that only bases in matching k-mers are counted, not the number of k-mers. When there are k+1 matching consecutive bases, two k-mers are generated. These are less 'valuable' than two non-overlapping k-mers. If $Count(d) \geq h$ for any diagonal band d, there is a seed hit for each position (i, j) in the band d unless position i was already in the range of a previously calculated local alignment.

The best values for h and s depend on things like ϵ and the read lengths.

For Daligner, the default k is 14, and assumed error rate is 0.85.

1.4 Seeding: implementation

Daligner is designed to use multiple threads and use the cache efficiently. Building and merge the lists in steps 1 and 3 is easy, since only one pass is needed for both actions. The elements of the lists are compressed into 64-bit integers. Daligner uses a radix sort [?][?] to sort the lists in steps 2 and 4. Each 64-bit integer is a vector of $P = \lceil hbits/B \rceil$, B-bit digits $(x_P, x_{P-1}, x, ..., x_1)$ where B is a parameter. The sort needs P passes, where each pass sorts the elements on a B-bit digit x_i. Each pass is done with a bucket sort [?] with 2^B buckets. Instead of a linked list, the integers in the list *src* are moved in precomputed segments $trg[bucket[b]...bucket[b+1]-1]$ of an array $trg[0...N-1]$ with the same size as *src*. For the p^{th} pass, $bucket[b] = \{i : src[i]_p < b\}$ for each $b \in [0, 2^B - 1]$.

```
for i = 0 to N-1 do
{
        b = src[i]_p
        trg[bucket[b]] = src[i]
        bucket[b] += 1
}
```

The algorithm takes $O(P(N + 2^B))$ time, but B and P are small fixed numbers so it is effectively $O(N)$. There are a lot of parallel sorting algorithms [?][?], but Daligner uses a new method that needs half the number of passes that traditional methods use. Each thread sorts a contiguous segment of size $part = \lceil N/T \rceil$ of *src* into *trg*, where T is the number of threads. This means each thread $t \in [0, T-1]$ has a bucket array $bucket[t]$ where $bucket[t][b] = \{i : src[i] < b$ or $src[i] = b$ and $i/part < t\}$. To reduce the number of passes, a bucket array for the next pass is filled during the current pass. Each thread counts the number of B-bit digits that will be handled in the next pass by itself and every other thread seperately. If the number at index i will be at index j and in bucket b next pass, then the count in the current pass must not be recorded for the thread

$i/part$ that currently sorts the number, but for thread $j/part$ that will sort it in the next pass. To do this we need to count the number of these events in $next[j/part][i/part][b]$ where $next$ is a $T \times T \times 2^B$ matrix. If $src[i]$ is about to be moved in the p^{th} pass, then $j = bucket[src[i]_p]$ and $b = src[i]_{p+1}$. This algorithm takes $O(N/T + T^2)$ time, assuming B and P are fixed.

```
int64 MASK = 2^B-1

sort_thread(int t, int bit, int N, int64 *src, int64 *trg, int *bucket, in
{
        for i = t*N to (t+1)*N-1 do
        {
                c = src[i]
                b = c >> bit
                x = bucket[b & MASK] += 1
                trg[x] = c
                next[x/N][(b >> B) & MASK] += 1
        }
}

int64 *radix_sort(int T, int N, int hbit, int64 src[0..N-1], int64 trg[0..
{
        int bucket[0..T-1][0..2^B-1], next[0..T-1][0..T-1][0..2^B-1]
        part = (N-1)/T + 1
        for l = 0 to hbit-1 in steps of B do
        {
                if (l != lbit)
                        bucket[t,b] = Sum_t next[u,t,b]
                else
                        bucket[t,b] = |{i : i/part == t and src[i] & MASK
                bucket[t,b] = Sum_u,(c<b) bucket[u,c] + Sum_(u<t) bucket[u
                next[u,t,b] = 0
                in parallel: sort_thread(t,l,part,src,trg,bucket[t],next[t
                (src,trg) = (trg,src)
        }
        return src
}
```

This algorithm is particularly cache efficient because each bucket sort uses two small arrays *bucket* and *next* that will usually fit in the L1 cache. Each bucket sort makes one sweep through *src* and 2^B sweeps through the bucket segments of *trg*. This totals $2^B + 1$ sweeps during each pass. Each sweep can be prefetched if it is small enough. This means a smaller B is better for caching behaviour, but this increases the number of passes $hbit/B$ that are needed. On most processors, e.g. an Intel i7, $B = 8$ gives the fastest radix sort [?]. The optimal number of threads is more complex, because they usually do not have their own caches.

1.5 Local Alignment

Assuming the filter finds a seed-hit $\rho = (i, j)$, the basic idea is compute furthest reaching waves in forward and reverse direction to find the alignment. The problem is that as the wave propagates further from ρ, it spans wider and wider since it occupies $2d + 1$ diagonals. The final alignment will have only one point from each wave so most of the points are wasted, but we only which ones until the whole alignment is done. Several strategies are used to trim the width of the wave by removing f.r. points that are unlikely to be in the final alignment.

One of the trimming strategies is stopping when a segment with very low correlation is found. This referred to as the *regional alignment quality.* F.r. points with less than $\mathcal{M}$ matches in the last C columns are removed. For example, when $\epsilon = .15$ then a segment with $M[k] < .55C = 33$ if $C = 60$ is probably not desirable.

To keep track of the matching/mismatching bases, we keep a bitvector $B(d, k)$ that represents the last $C = 60$ columns of the best path from ρ to a given f.r. point $F(d, k)$. A 0 will denote a mismatch and a 1 a match. This is easily updated by left-shifting a 0 or 1. The number of matches $M(d, k)$ can be tracked by observing the bit that is shifted out. Listing 1.1 shows pseudo-code that computes $W_\rho(d + 1)[low - 1, hgh + 1]$ from $W_\rho(d)[low, hgh]$ assuming $[low, hgh] \subseteq [\kappa - d, \kappa + d]$ is the trimmed result of the wave $W_\rho(d)$. Note that the array W only holds the B-coordinate j of each f.r. point (i, j), since $i = j + k$.

Listing 1.1: Local Alignment

```
MASKC  = 1 << (C-1)
W[low-2] = W[hgh+2] = W[hgh+1] = y = yp = -1
for k = low-1 to hgh+1 do
{
        (ym,y,yp) = (y,yp+1,W[d+1]+1)
        if (ym = min(ym,y,yp))
                (y,m,b) = (ym,M[k-1],B[k-1])
        else if (yp = min(ym,y,yp)
                (y,m,b) = (yp,M[k+1],B[k+1])
        else
                (y,m,b) = (y,M[k],B[k])

        if (b & MASKC != 0)
                m -= 1
        b <<= 1
        while (B[y] == A[y+k])
        {
                y += 1
                if (b & MASKC == 0)
                        m += 1
                b = (b << 1) | 1
        }
        (W[k],M[k],B[k]) = (y,m,b)
}
```

The second trimming principle involves only keeping f.r. points which are within $\mathcal{L}$ anti-diagonals of the maximal anti-diagonal reached by its wave. It makes sense that the f.r. point on diagonal k^* that will be in the final alignment is on a greater anti-diagonal $i+k$ than points that are not. As the other f.r. points in the wave move away from diagonal k^*, their anti-diagonal values decrease rapidly, and the wave gets the appearance of an arrowhead. The higher the correlation of the alignment, the sharper the arrowhead becomes. This means that points far enough behind the tip can be almost certainly removed. A value of $\mathcal{L} = 30$ is a universally good value for trimming [**?**]. Formally, for each wave computed from the previous trimmed wave, each f.r. point from $[low-1, hgh+1]$ that has either $M[j] < \mathcal{M}$ or $(2W[k^*]+k^*)-(2W[j]+j) > \mathcal{L}$ is removed. Note that $W[k]$ contains the B-coordinate j, and $k = i - j$, so $(2W[k] + k) = i + j$.

The Daligner paper does not give a formal proof, but an argument why the average width of the wave $hgh - low$ is constant for any fixed value of ϵ. This means that the alignment finding algorithm has linear expected time as a function of alignment length. Consider two f.r. points A and B, where A lies on an alignment path with correlation $1 - 2\epsilon$ or higher, and B does not. For the next wave, point A moves forward with one difference and then slides on average $\alpha = (1-\epsilon)^2/(1-(1-\epsilon)^2)$ bases. Point B has no such correlation, so it jumps one difference and then only slides $\beta = 1/(\Sigma - 1)$ bases, assuming each base is equally likely. So an f.r. point d diagonals away from the final path has involved d jumps off the path, and is on average $d(\alpha - \beta)$ anti-diagonals behind the best f.r. point in the wave. The average width of a wave with an $\mathcal{L}$ lag cutoff is less than $2\mathcal{L}/(\alpha - \beta)$. This last step is incorrect because the statistics of average random path length under this difference model is complexer than assuming all random steps are the same. However, there is a definite expected value of path length with d differences, so the basis of the argument holds, although with a different value for β. When ϵ goes to 0, there is a very long slide from the starting point ρ. Each f.r. point not on the path should lag a lot behind the best f.r. point. As α increases as ϵ goes to 0, this explains further why the wave becomes very pointy and narrow.

The alignment finding algorithm ends because either the boundary of the edit graph is reached, or because all f.r. points have failed the regional alignment criterion, this means that the reads are probably not correlated anymore. In the second case, the best point in the last wave should not be reported as endpoint of the alignment, because the last columns could all be mismatches. Because the overall path should have an average correlation of $1 - 2\epsilon$, only a polished point with greatest anti-diagonal can be the end of a path. A *polished point* is a point for which the last $E \leq C$ columns are such that every suffix of the last E columns have a correlation of $1 - 2\epsilon$ or better, this is called being *suffix positive*. Daligner keep track of the polished f.r. point with the greatest anti-diagonal during the computation of the waves, it does so by testing if each bit-vector of the leading f.r. points is suffix positive. Testing bit-vector e can be done in $O(1)$ time by precomputing a table $SP[e]$ with 2^E elements. Define $Score(\emptyset) = 0$ and recursively $Score(1b) = Score(b) + \alpha$ and $Score(0b) = Score(b) - \beta$ where $\alpha = 2\epsilon$ and $\beta = 1 - 2\epsilon$. Note that if bit-vector b has m matches and d differences, then $Score(b) = \alpha m - \beta d$. If this is non-negative then $m/(m+d) \geq 1 - 2\epsilon$, which means b has a correlation of $1 - 2\epsilon$

or higher. Now let $SP[e] = min\{Score(b) : b \text{ is a suffix of } e\}$. The table SP can be built in linear time by computing $Score$ over the trie [?][?] of all E-bit vectors and taking the minimum of each path of the trie.

However for large values of E, say 30, the table SP gets too big. To solve this, a size D, say 15, is chosen for which the SP table is reasonable. Consider an E-bit vector e that consists of $X = E/D$, D-bit segments $e_X \cdot e_{X-1} \cdot ... \cdot e_1$. Precompute table SP as before, but now only for D bits, and a table SC for D-bit vectors as well, where $SC[b] = Score(b)$. With these two 2^D tables it takes $O(X)$ time to determine if the longer bit-vector e is suffix positive by calculating if $Polish(X)$ is true with the following recurrences:

$$Score(x) = \begin{cases} Score(x-1) + SC[e_x] & \text{if } x \geq 1 \\ 0 & \text{if } x = 0 \end{cases} \tag{1.2}$$

$$Polish(x) = \begin{cases} Polish(x-1) \text{ and } Score(x-1) + SP[e_x] \geq 0 & \text{if } x \geq 1 \\ true & \text{if } x = 0 \end{cases} \tag{1.3}$$