Efficient Local Alignment Discovery amongst Noisy Long Reads

Gene Myers[1] myers@mpi-cbg.de *

MPI for Molecular Cell Biology and Genetics, 01307 Dresden, GERMANY

Abstract. Long read sequencers portend the possibility of producing reference quality genomes not only because the reads are long, but also because sequencing errors and read sampling are almost perfectly random. However, the error rates are as high as 15%, necessitating an efficient algorithm for finding local alignments between reads at a 30% difference rate, a level that current algorithm designs cannot handle or handle inefficiently. In this paper we present a very efficient yet highly sensitive, threaded *filter*, based on a novel *sort and merge* paradigm, that proposes *seed points* between pairs of reads that are likely to have a significant local alignment passing through them. We also present a *linear* expected-time heuristic based on the classic $O(nd)$ difference algorithm [1] that finds a local alignment passing through a seed point that is exceedingly sensitive, failing but once every billion base pairs. These two results have been combined into a software program we call `DALIGN` that realizes the fastest program to date for finding overlaps and local alignments in very noisy long read DNA sequencing data sets and is thus a prelude to *de novo* long read assembly

1 Introduction & Summary

The PacBio RS II sequencer is the first operational "long read" DNA sequencer [2]. While its error rate is relatively high (ϵ = 12-15% error), it has two incredibly powerful offsetting properties, namely, that (a) the set of reads produced is a nearly Poisson sampling of the underlying genome, and (b) the location of errors within reads is truly randomly distributed. Property (a), by the Poisson theory of Lander and Waterman [3], implies that for any minimum target coverage level k, there exists a level of sequencing coverage c that guarantees that every region of the underlying genome is covered k times. Property (b), from the early work of Churchill and Waterman [4], implies that the accuracy of the consensus sequence of k such sequences is $O(\epsilon^k)$ which goes to 0 as k increases. Therefore, provided the reads are long enough that repetitive genome elements do not confound assembling them, then in principle a (near) perfect *de novo* reconstruction of a genome at any level of accuracy is possible given *enough* coverage c.

These properties of the reads are in stark contrast to those of existing technologies where neither property is true. All previous technologies make reproducible sequencing errors. A typical rate of occurrence for these errors is about 10^{-4} implying at best a Q40 reconstruction is possible, whereas in principle any desired reconstruction accuracy is possible with the long reads, e.g., a Q60 reconstruction has been demonstrated for *E. coli* [5]. All earlier technologies also exhibit clear sampling biases, typically due to a biased amplification or selection step, implying that many regions of a target genome are not sequenced. For example, some PCR-based instruments often fail to sequence GC rich stretches. So because their error and sampling are unbiased, the new long read technologies are poised to enable a dramatic shift in the state of the art of *de novo* DNA sequencing.

The questions then are (a) what level of coverage c is required for great assembly, i.e. how cost-effectively can one get near the theoretical ideal above, and (b) how does one build an assembler that works with such high error rates and long reads? The second question is important because most current assemblers do not work on such data as they assume much lower error rates and much shorter reads, e.g. error rates less than 2% and read lengths of 100-250bp. Moreover, the algorithms within these assemblers are specifically tuned for these operating points and some approaches, such as the de-Bruijn graph [6] would catastrophically fail at rates over 10%.

Finding overlaps is typically the first step in an overlap-layout-consensus (OLC) assembler design [7] and is the efficiency bottleneck for such assemblers. In this paper, we develop an efficient algorithm and software for finding all significant local alignments between reads in the presence of the high error rates of the long reads. Finding local alignments is more general then finding overlaps, and we do so because it allows us to find repeats, chimers, undetected vector sequence and other artifacts that must be detected in order to

* Supported by the Klaus Tschira Stiftung, Heidelberg, GERMANY

achieve near perfect assemblies. To this authors knowledge, the only previous algorithm and software that can effectively accommodate the level of error in question is BLASR [8] which was original designed as a tool to map long reads to a reference genome, but can also be used for the assembly problem. Empirically our program, DALIGN, is more sensitive while being typically 20 to 40 times faster depending on the data set.

We make use of the same basic filtration concept as BLASR, but realize it with a series of highly optimized threaded radix sorts (as opposed to a BWT index [9]). While we did not make a direct comparison here, we believe the cache coherence and thread ability of the simpler sorting approach is more time efficient then using a more sophisticated but cache incoherent data structure such as a Suffix Array or BWT index. But the real challenge is improving the speed of finding local alignments at a 30-40% difference rate about a seed hit from the filter, as this step consumes the majority of the time, e.g. 85% or more in the case of DALIGN. To find overlaps about a seed hit, we use a novel method of adaptively computing furthest reaching waves of the classic $O(nd)$ algorithm [1] augmented with information that describes the match structure of the last p columns of the alignment leading to a given furthest reaching point. Each wave on average contains a small number of points, e.g. 8, so that in effect an alignment is detected in time linear in the number of columns in the alignment.

In practice DALIGN achieved the following CPU and wall-clock times on 3 publicly available PacBio data sets [10]. The 48X *E coli* data set can be compared against itself in less than 5.4 wall clock minutes on a Macbook Pro with 16Gb of memory and a 4-core i7-processor. The 89X Arabadopsis data set can be processed in 71 CPU hours or 10 wall-clock minutes on our modest 480 core HPC cluster, where each node is a pair of 6-core Intel Xeon E5-2640's at 2.5GHz with 128Gb (albeit only 50-60Gb is used per node). Finally on the 54X human genome data set, 15,600 CPU hours or 32-33 wall-clock hours are needed. Thus our algorithm and software enables the assembly of gigabase genomes in a "reasonable" amount of compute time (e.g., compared to the 404,000 CPU hours reported for BLASR).

2 Preliminaries: Edit Graphs, Alignments, Paths, and F.R.-Waves

DALIGN takes as input a block $\mathcal{A}$ of M long reads $A^1, A^2, \ldots A^M$ and another block $\mathcal{B}$ of N long reads $B^1, B^2, \ldots B^N$ over alphabet $\Sigma = 4$, and seeks read subset pairs $P = (a, i, g) \times (b, j, h)$ 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 we call τ the minimum alignment length and ϵ the average error rate. We further will speak of $1 - 2\epsilon$ as the *correlation* or *percent identity* of the alignment. It will also be convenient throughout to introduce $\Sigma_{\mathcal{A}} = \sum_{a=1}^{M} |A^a|$, the total number of base pairs in $\mathcal{A}$, and $\max_{\mathcal{A}} = \max_a |A^a|$ the length of the longest read in $\mathcal{A}$.

Most readers will recall that an *edit graph* for read $A = a_1 a_2 \ldots a_m$ versus $B = b_1 b_2 \ldots b_n$ is a graph with an $(m + 1) \times (n + 1)$ array of vertices (i, j) ϵ $[0, M] \times [0, N]$ and the following edges:

(a) *deletion* edges $(i - 1, j) \rightarrow (i, j)$ with label $\begin{bmatrix} a_i \\ - \end{bmatrix}$ if $i > 0$.

(b) *insertion* edges $(i, j - 1) \rightarrow (i, j)$ with label $\begin{bmatrix} - \\ b_j \end{bmatrix}$ if $j > 0$.

(c) *diagonal* edges $(i - 1, j - 1) \rightarrow (i, j)$ with label $\begin{bmatrix} a_i \\ b_j \end{bmatrix}$ if $i, j > 0$

A simple exercise in induction reveals that the sequence of labels on a path from (i, j) to (g, h) in the edit graph spells out an alignment between $A[i + 1, g]$ and $B[j + 1, h]$. Let a *match* edge be a diagonal edge for which $a_i = b_j$ and otherwise call the diagonal edge a *substitution* edge. Then if match edges have weight 0 and all other edges have weight 1, it follows that the weight of a path is the number of differences in the alignment it models. So our goal in edit graph terms is to find read subset pairs P such that $len(P) \geq \tau$ and the lowest scoring path between (i, j) and (g, h) in the edit graph of A^a versus B^b has cost no more than $2\epsilon \cdot len(P)$.

In 1986 we presented a simple $O(nd)$ algorithm [1] for comparing two sequences that centered on the idea of computing progressive "waves" of *furthest reaching* (f.r.) points. Starting from a point $\rho = (i, j)$ in *diagonal* $\kappa = i - j$ of the edit graph of two sequences, the goal is to find the longest possible paths starting at ρ, first with 0-differences, then with 1-differences, 2-differences, and so on. Note carefully that after d

differences, the possible paths can end in diagonals $\kappa \pm d$. In each of these $2d+1$ diagonals we want to know the furthest point on the diagonal that can be reached from ρ with exactly d differences which we denote by $F_\rho(d,k)$. We call these points collectively the d-wave emanating from ρ and formally $W_\rho(d) = \{F_\rho(d,\kappa-d), \ldots F_\rho(d,\kappa+d)\}$. We will more briefly refer to $F_\rho(d,k)$ as the f.r. d-point on k where ρ will be implicit understood from context. In the 1986 paper we proved that:

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

where $Slide(k,(i,j)) = (i,j) + max\{\Delta : a_{i+1}a_{i+2}\ldots a_{i+\Delta} = b_{j+1}b_{j+2}\ldots b_{j+\Delta}\}$. In words, the f.r. d-point on k can be computed by first finding the furthest of (a) the f.r. $(d-1)$-point on $k-1$ followed by an insertion, or (b) the f.r. $(d-1)$-point on k followed by a substitution, or (c) the f.r. $(d-1)$-point on $k+1$ followed by a deletion, and thereafter progressing as far as possible along match edges (a "slide"). Formally a point (i,j) is furthest if its anti-diagonal, $i+j$, is greatest. Next, it follows easily that the best alignment between A and B is the smallest d such that $(m,n) \in W_{(0,0)}(d)$ where m and n are the length of A and B, respectively. So the $O(nd)$ algorithm simply computes d-waves from $(0,0)$ in order of d until the goal point (m,n) is reached in the d^{th} wave. It can further be shown that the expected complexity is actually $O(n+d^2)$ under the assumption that A and B are non-repetiitve sequences. In what follows we will be computing waves adaptively and in both the forward direction, as just described, and in the reverse direction, which is conceptually simply a matter of reversing the direction of the edges in the edit graph.

3 Rapid Seed Detection: Concept

Given blocks $\mathcal{A}$ and $\mathcal{B}$ of long, noisy reads, we seek to find local alignments between reads that are sufficiently long (parameter τ) and sufficiently stringent (parameter ϵ). For our application ϵ is much larger than typically contemplated in prior work, 10-15%, but the reads are very long, 10Kbp, so τ is large, 1 or 2Kbp. Here we build a filter that eliminates read pairs that cannot possibly contain a local alignment of length τ or more, by counting the number of conserved k-mers between the reads. A careful and detailed analysis of the statistics of conserved k-mers in the operating range of ϵ and τ required by long read data, has previously been given in the paper about the `BLASR` program [8]. So here we just illustrate the idea by giving a rough estimate assuming all k-mer matches are independent events. Under this simplifying assumption, it follows that a given k-mer is conserved with probability $\pi = (1-2\epsilon)^k$ and the number of conserved k-mers in an alignment of τ base pairs is roughly a Bernouilli distribution with rate π and thus an average of $\tau \cdot \pi$ conserved k-mers are expected between two reads that share a local alignment of length τ. As an example, if $k=14$, $\epsilon = 15\%$, and $\tau = 1500$, then $\pi \approx .7^{14} = .0067$, we expect to have 10.0 14-mers conserved on average, and only .046% of the sought read pairs have 1 or fewer hits between them and only .26% have 2 or fewer hits. Thus a filter with expected sensitivity 99.74% examines only read pairs that have 3 or more conserved 14-mers. `BLASR` and `DALIGN` effectively use this strategy where one controls sensitivity and specificity by selecting k and the number of k-mers that must be found. Beyond this point our methods are completely different.

First, we improve specificity by (a) computing the number of conserved k-mers in bands of diagonals of width 2^s between two reads (as opposed to the entire reads) where a typical choice for s is 6, and (b) thresholding a hit on the number of bases, h, in conserved k-mers (as opposed to the number of k-mers). Condition (a) increases specificity as it limits the set of k-mers to be counted at a potentially slight loss of sensitivity because an alignment can have an insertion or deletion bias and so can drift across bands rather than staying in a single band. To understand condition (b) note that 3 consecutive matching k-mers involve a total of $k+2$ matching bases, whereas 3 disjoint matching k-mers involve a total of $3k$ matching bases. Under our simplifying assumption the first situation happens with probability $\pi^{1+2/k}$ and the second with probability π^3, i.e. one is much more specific than the other. By counting the number of bases involved in k-mer hits we ensure that all filters hits have roughly the same statistical frequency.

There are many ways to find matching k-mers over an alphabet Σ, specifically of size 4 in this work, most involving indices such as Suffix Arrays [11] or BWT indices [9]. We have found in practice that a much simpler series of highly optimized sorts can similarly deliver the number of bases in k-mers in a given diagonal band between two reads. Given blocks $\mathcal{A}$ and $\mathcal{B}$ we proceed as follows:

1. Build the list $List_A = \{(kmer(A^a, i), a, i)\}_{a,i}$ of all k-mers of the $\mathcal{A}$ block and their positions, where $kmer(R, i)$ is the k-mer, $R[i-k+1, i]$.
2. Similarly build the list $List_B = \{(kmer(B^b, j), b, j)\}_{b,j}$.
3. Sort both lists in order of their k-mers.
4. In a merge sweep of the two k-mer sorted lists 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.
5. Sort $List_M$ lexicographically on a, b, and i where a is most significant and i least.

To keep the following analysis simple, let us assume that the sizes of the two blocks are both roughly the same, say N. Steps 1 and 2 are easily seen to take $O(N)$ time and space. The sorts of steps 3 and 5 are in theory $O(LlogL)$ where L is the list size. The only remaining complexity question is how large is $List_M$. First note that there is a contribution (i) from k-mers that are purely random chance, and (ii) from conserved k-mers that are due to the reads actually being correlated. The first term is N^2/Σ^k as we expect to see a given k-mer N/Σ^k times in each block. For case (ii), suppose that the data set is a c-fold covering of an underlying genome, and, in the worst case, the $\mathcal{A}$ and $\mathcal{B}$ blocks are the same block and contain all the data. The genome is then of size N/c and each position of the genome is covered by c reads by construction. Because c/π k-mers are on average conserved amongst the c reads covering a given position, there are thus $N/c \cdot (c/\pi)^2 = (Nc/\pi^2)$ matching k-mer pairs by non-random correlations. In most projects c is typically 50-100 whereas π is typically 1/100 (e.g. $k = 14$ and $\epsilon = 15\%$) implying somewhat counter-intuitively that the non-random contribution is dominated by the random contributions! Thus $List_M$ is $O(N^2/\Sigma^k)$ in size and so in expectation the time for the entire procedure is dominated by Step 5 which takes $O(N^2 logN/\Sigma^k)$. Finally, suppose the total amount of data is M and we divide it into blocks of size Σ^k all of which are compared against each other. Then the time for each block comparison is $O(k\Sigma^k)$ using $O(\Sigma^k)$ space, that is linear time and space in the block size. Finally, there are M/Σ^k blocks implying the total time for comparing all blocks is $O(kM \cdot (M/\Sigma^k))$. So our filter, like all others, still has a quadratic component in terms of the number of occurrences of a given k-mer in a data set. With linear times indices such as BWT's the time can theoretically be improved by a factor of k. However, in practice the k arises from a radix sort that actually makes only $k/4$ passes and is so highly optimized, threaded, and cache coherent that we believe it likely outperforms a BWT approach by a considerable margin. At the current time all we can say is that DALIGN which *includes alignment finding* is 20-40 times faster than BLASR which uses a BWT (see Table 6).

For the sorted list $List_M$, note that all entries involving a given read pair (a, b) are in a single contiguous segment of the list after the sort in Step 5. Given parameters h and s, for each pair in such a segment, we place each entry (a, b, i, j) in both *diagonal bands* $d = \lfloor (i-j)/2^s \rfloor$ and $d+1$, and then determine the number of bases in the A-read covered by k-mers in each pair of bands diagonal band, i.e. $Count(a, b, d) = |\cup \{w(A^a, a, i) : (a, b, i, j) \in List_M \text{ and } \lfloor (i-j)/2^s \rfloor = d \text{ or } d+1\}|$. Doing so is easy in linear time in the number of relevant entries as they are sorted on i. If $Count(a, b, d) \geq h$ then we have a hit and we call our local alignment finding algorithm to be described, with each position (i, j) in the bucket d *unless* the position i is already within the range of a local alignment found with an index pair searched before it. This completes the description of our filtration strategy and we now turn to its efficient realization.

4 Rapid Seed Detection: Algorithm Engineering

Todays processors have multiple cores and typically a 3-level cache hierarchy implying memory fetch times vary by up to 100 for L1 cache hits versus a miss at all three cache levels. We therefore seek a realization of the algorithm above that is parallel over T threads and is cache coherent. Doing so is easy for steps 1, 2, and 4 and we optimize the encoding of the lists by squeezing their elements into 64-bit integers. So the key problem addressed in the remainder of this section is how to realize a threadable, memory coherent sort of an array $src[0..N-1]$ of N 64-bit integers in steps 3 and 5.

We chose a *radix sort* [12] where each number is considered as a vector of $P = \lceil hbits/B \rceil$, B-bit digits, $(x_P, x_{P-1}, \ldots, x_1)$ and B is a free parameter to be optimally chosen empirically later. A radix sort sorts the numbers by *stably* sorting the array on the first B-bit digit x_1, then on the second x_2, and so on to x_P in P sorting passes. Each B-bit sort is achieved with a *bucket sort* [12] with 2^B buckets. Often this basic sort is realized with a linked list, but a much better strategy sequentially moves the integers in src into pre-computed segments, $trg[\, bucket[b] \,..\, bucket[b+1]-1\,]$ of an auxiliary array $trg[0..N-1]$ where, for the p^{th} pass, $bucket[b] = \{i : src[i]_p < b\}$ for each $b \in [0, 2^B - 1]$. In code, the p^{th} bucket sort is:

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

Asymptotically the algorithm takes $O(P(N+2^B))$ time but B and P are fixed small numbers so the algorithm is effectively $O(N)$.

While papers on threaded sorts are abundant [13], we never the less present our pragmatic implementation of a threaded radix sort, because it uses half the number of passes over the array that other methods use, and accomplishing this is non-trivial as follows. In order to exploit the parallelism of T threads, we let each thread sort a contiguous segment of size $part = \lceil N/T \rceil$ of the array src into the appropriate locations of trg. This requires that each thread $t \in [0, T-1]$ has its own bucket array $bucket[t]$ where now $bucket[t][b] = \{i : src[i] < b \text{ or } src[i] = b \text{ and } i/part < t\}$. In order to reduce *the number of sweeps over the arrays by half*, we produce the bucket array for the *next* pass while performing the current pass. But this is a bit complex because each thread must count the number of B-bit numbers in the next pass that will be handled by not only itself but every other thread separately! That is, if the number at index i will be at index j and bucket b in the next pass then the count in the current pass must be recorded not for the thread $i/part$ currently sorting the number, but for the thread $j/part$ that will sort the number in the next pass. To do so requires that we actually count the number of such events in $next[j/part][i/part][b]$ where now $next$ is a $T \times T \times 2^B$ array. It remains to note that when $src[i]$ is about to be moved in the p^{th} pass, then $j = bucket[src[i]_p]$ and $b = src[i]_{p+1}$. The complete algorithm is presented below in C-style pseudo-code where unbound variables are assumed to vary over the range of the variable. It is easily seen to take $O(N/T + T^2)$ 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, int *next)
{ 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..N-1])
{ 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 == b  } |
      bucket[t,b] = Sum_u,(c<b) bucket[u,c] + Sum_(u<t) bucket[u,b]
      next[u,t,b] = 0
      in parallel: sort_thread(t,l,part,src,trg,bucket[t],next[t]])
      (src,trg) = (trg,src)
    }
  return src
}
```

We conclude by emphasizing why this approach to sorting is a particularly efficient realization of a very large array sort. Each bucket sort involves two small arrays *bucket* and *next* that will typically fit in the fastest L1 cache. Each bucket sort makes a single *sweep* through *src* while making 2^B sweeps through the bucket segments of *trg*. Thus $2^B + 1$ cache-coherent sweeps occur during each bucket sort pass. Each sweep

can be prefetched as long as their number does not exceed the interleaving of the cache architecture. So the smaller B is the better the caching and prefetching behavior will be, but this is counter balanced by the increasing number of passes $hbit/B$ that are required. We found that on most processors, e.g. an Intel i7, the minimum total time for our radix sort occurs with $B = 8$ which conveniently is the number of bits in a byte.

The number of threads T to employ is a complex question despite the fact that there is no communication or synchronization required between threads in our algorithm and the non-parallel overhead is only $O(T^2)$. The reason is that every thread does not have its own set of caches. They generally have an independent L1 cache, but then share the L2 and L3 caches. This means that the actual number of sweeps taking place is $T(2^B + 1)$ which at the level of the L2 cache begins to induce interleaving interference. Nonetheless, speed up is still very good. For example, on a 4-core Intel i7, the speedup achieved was 3.6 !

5 Rapid Local Alignment Discovery from a Seed

We now turn to finding local alignments of length τ or more and correlation $1 - 2\epsilon$ or better given a seed-hit (i, j) between two reads A and B reported by the filter above. The basic idea is to compute f.r. waves in both the forward and reverse direction from the seed point $\rho = (i, j)$ (see Section 2). The problem of course is that the d-wave from ρ spans $2d+1$ diagonals, that is, waves become wider and wider as one progresses away from ρ in each direction. We know that only one point in each wave will actually be in the local alignment ultimately reported, but we only know these points after all the relevant waves have been computed. We use several strategies to trim the span of a wave by removing f.r. points that are extremely unlikely to be in the desired local alignment.

A key idea is that a desired local alignment should not over any reasonable segment have an exceedingly low correlation. To this end imagine keeping a bit vector $B(d, k)$ that actually models the last, say $C = 60$ columns, of the best path/alignment from ρ to a given f.r. point $F(d, k)$ in the d-wave. That is a 0 will denote a mismatch in a column of the alignment and a 1 will denote a match. This is actually relatively easy to do: left-shift in a 0 when taking an indel or substitution edge and then left-shift in a 1 with each matching edge of a snake. One can further keep track of exactly how many matches $M(d, k)$ there are in the alignment by observing the bit that gets shifted out when a new bit is shifted in. The pseudo-code below computes $W_\rho(d+1)[low - 1, hgh + 1]$ from $W_\rho(d)[low, hgh]$ assuming that $[low, hgh] \subseteq [\kappa - d, \kappa + d]$ is the interval of $W_\rho(d)$ that we have decided to retain (to be described below). Note that the code computes the information for each wave in place within the arrays `W`, `B`, and `M` where `W` simply records the B-coordinate, j, of each f.r. point (i, j) as we know the diagonal k of the point, and hence that $i = j + k$.

```
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)
  }
```

A very simple principle for *trimming* a wave is to remove f.r. points for which the last C columns of the alignment have less than say $\mathcal{M}$ matches, we call this the *regional alignment quality*. For example, if $\epsilon = .15$ then one almost certainly does not want a local alignment that contains a C column segment for which $\mathtt{M}[\mathtt{k}] < .55C = 33$ if $C = 60$. A second trimming principle is to keep only f.r. points which are within $\mathcal{L}$ anti-diagonals of the maximal anti-diagonal reached by its wave. Intuitively, the f.r. point (i, j) on diagonal $k^\star$ *on the desired path* is on a greater anti-diagonal $i + j$ than those of the points on either side of it in the same wave, and as one progresses away from diagonal $k^\star$, the anti-diagonal values of the wave recede rapidly, giving the wave the appearance of an arrowhead. The higher the correlation rate of the alignment, the sharper the arrow head becomes and the points far enough behind the tip of the arrow are almost certainly not points on an optimal local alignment. So for each portion of a wave computed from the previous trimmed wave, we trim away f.r. points from $[low - 1, hgh + 1]$ that either have $\mathtt{M}[j] < \mathcal{M}$ or $(2\mathtt{W}[k^\star] + k^\star) - (2\mathtt{W}[j] + j) > \mathcal{L}$. In the experimental section we show that $\mathcal{L} = 30$ is a universally good value for trimming.

While not a formal proof per se, the following argument explains why in the empirical results section we see that the average wave size $hgh - low$ is a constant for any fixed value of ϵ, and hence why the alignment finding algorithm is linear expected time in the alignment length. Imagine the extension of an f.r. point that is actually on the path of an alignment with correlation $1 - 2\epsilon$ or better. For the next wave, this point jumps forward one difference and then "slides" on average $\alpha = (1 - \epsilon)^2/(1 - (1 - \epsilon)^2)$ matching diagonals. Contrast this to an f.r. point off the alignment path which jumps one difference and then only slides $\beta = 1/(\Sigma - 1)$ diagonals, assuming every base is equally likely. On average then, an entry d diagonals away from the alignment path, has involved d jumps from f.r. points off the path, and hence is $d(\alpha - \beta)$ behind the f.r. point on the alignment path in the same wave. Thus the average width of a wave trimmed with lag cutoff $\mathcal{L}$ would be less than $2\mathcal{L}/(\alpha - \beta)$. This last step of the argument is incorrect as the statistics of average random path length under the difference model is more complex then assuming all random steps are the same, but there is a definite expected value of path length with d-differences, and therefore the basis of the argument holds, albeit with a different value for β. Since α increases as ϵ decreases, it further explains why the wave becomes more pointy and narrower as ϵ goes to zero.

The computation of successive waves eventually ends because either (a) the boundary of the edit graph of A and B is reached, or (b) all the f.r. points fail the regional alignment quality criterion in which case one can assume that the two reads no longer correlate with each other. In case (b), one should not report the best point in the last wave, as the trimming criterion is overly permissive (e.g. the last 5 columns could all be mismatches!) Because we seek alignments that have an average correlation rate of $1 - 2\epsilon$, we choose to end the path at a *polished point* with greatest anti-diagonal 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. We call such alignments *suffix positive* (at rate ϵ) for reasons that will become obvious momentarily. We must then keep track of the polished f.r. point with greatest anti-diagonal as the waves are computed, which in turns means that we must test the alignment bit-vector of the leading f.r. point(s) for the suffix positive property in each wave.

One can in $O(1)$ time determine if an alignment bit-vector e is suffix positive by building a 2^E-element table $SP[e]$ as follows. Let $Score(\in) = 0$ and recursively let $Score(1b) = Score(b) + \alpha$ 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 $Score(b) = \alpha m - \beta d$ and if this is non-negative then it implies that $m/(m + d) \geq 1 - 2\epsilon$, i.e. b's alignment has correlation $1 - 2\epsilon$ or better. Let $SP[e] = min\{Score(b) : b \text{ is a suffix of } e\}$. Clearly $SP[e] \geq 0$ if and only if e is suffix positive (at rate ϵ). By computing *Score* over the trie of all length E bit vectors and recording the minimum along each path of the trie, the table SP can be built in linear time.

However if E is large, say 30 (as we generally prefer to set it), then the table gets too big. If so, then pick a size D (say 15) for which the SP-table size is reasonable and consider an E-bit vector e to consist of $X = E/D$, D-bit segments $e_X \cdot e_{X-1} \cdot \ldots \cdot e_1$. Precompute the table SP, but for only D bits, and a table SC for bit-vectors of the same size where $SC[b] = Score(b)$. Given these two 2^D tables one can then determine if the longer bit-vector e is suffix positive in $O(X)$ time by calculating whether $Polish(X)$ is true or not with the following recurrences:

$$\begin{aligned} Score(x) &= \begin{cases} Score(x-1) + SC[e_x] & \text{if } x \geq 1 \\ 0 & \text{if } x = 0 \end{cases} \\ Polish(x) &= \begin{cases} Polish(x-1) \text{ and } Score(x-1) + SP[e_x] \geq 0 & \text{if } x \geq 1 \\ true & \text{if } x = 0 \end{cases} \end{aligned} \tag{2}$$

In summary, we compute waves of f.r. points keeping only those that are locally part of a good alignment and not too far behind the leading f.r. point. The waves stop either when a boundary is reached, in which case the boundary point is taken as the end of the alignment, or all possible points are eliminated, in which case the furthest polished f.r. point is taken as the end of the alignment (in the given direction). The search takes place both in the forward direction and the reverse direction from a seed tip ρ. The intervals of A and B at which the forward and reverse searches end is reported as a local alignment if the alignment has length τ or more.

Clearly the algorithm is heuristic: (a) it could fail to find an alignment by virtue of eliminating incorrectly an f.r. point on the alignment, and (b) it could over report alignments whose correlation is less than $1-2\epsilon$ as local segments of worse quality are permitted depending on the setting of $\mathcal{M}$. We will examine the sensitivity and specificity of the algorithm in the Empirical Performance section, but for the moment indicate that with reasonable choices of $\mathcal{M}$ and $\mathcal{L}$ the algorithm fails less, than once in a billion base pairs, i.e. (a) almost never happens. It is our belief that this heuristic variation of the $O(nd)$ algorithm is superior to any other filter verification approach for local alignments in the case of identity matching over DNA while simultaneously being extremely sensitive. Intuitively this is because the heuristic explores many fewer vertices of the edit graph than dynamic programming based approaches because in expectation the span $hgh - low$ of trimmed waves is a small constant, that is, an alignment is found in linear expected time with near certainty.

6 Empirical Performance

All trials reported in this section were run on a Macbook Pro with a 2.7GHz Intel Core i7 and the code was compiled with gcc version 4.2.1 with the -O4 level of optimization set.

For a given setting of ϵ, we ran trials to determine the sensitivity of the local alignment algorithm in terms of the trimming parameters $\mathcal{M}$ and $\mathcal{L}$. Each trial consisted of generating a 1Mbp random DNA sequence (with every base equally likely) and then peppering in random differences at rate ϵ into two distinct copies. The two perturbed copies were then given to the wave algorithm with seed point $(0,0)$. For various settings of the trimming parameters and ϵ we ran 1000 trials and recorded (a) what fraction of the trials were *successful* in that the entire 1Mbp alignment between the two copies was reported (Table 2), (b) the average wave span (Table 3), and (c) the time taken.

Perturbation (ϵ)	Observed Correlation	Effective Perturbation
15.0%	76.1%	12.45%
10.0%	82.8%	8.60%
5.0%	90.7%	4.35%
2.5%	95.2%	2.40%
1.0%	98.0%	1.00%

Table 1. Perturbation versus Observed Correlation and Effective Perturbation

The first thing we observed was that the perturbed copies of a sequence actually aligned with much better correlation than $1-2\epsilon$ and the larger ϵ the larger the relative improvement. We thus define the *effective perturbation* as the value $\epsilon^\star$ such that $1-2\epsilon^\star$ equals the observed correlation. Table 1 gives the observed correlation and effective perturbation for a range of values of ϵ.

The success rate and wave span both increase monotonically as $\mathcal{L}$ increases and as $\mathcal{M}$ decreases. In Table 2, we observe that achieving a 100% success rate depends very crucially on $\mathcal{M}$ being small enough, e.g. $\mathcal{M}$ must be 55% or less when the perturbation is $\epsilon = 15\%$, 60% or less for $\epsilon = 10\%$, and so on. But one should further note in Table 3 that the average wave span is virtually independent of $\mathcal{M}$ and really depends only on $\mathcal{L}$, at least for the values of $\mathcal{M}$ that are required to have a 100% success rate. One might then think that only the lag threshold is important and trimming on $\mathcal{M}$ can be dropped, but one must remember that in the general case, when two sequences stop aligning, it is regional alignment quality that stops the extension beyond the end of the local alignment.

		$\mathcal{L}$							
$1-2\epsilon$	$\mathcal{M}$	15	20	25	30	35	40	45	50
70%	55%	0.68	0.97	1.00	1.00	1.00	1.00	1.00	1.00
	60%	0.27	0.66	0.74	0.74	0.74	0.74	0.73	0.72
	65%	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
80%	55%	0.99	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	60%	0.96	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	65%	0.86	0.93	0.94	0.92	0.94	0.95	0.95	0.94
	70%	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
90%	70%	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	75%	0.91	0.92	0.92	0.92	0.94	0.94	0.94	0.94
	80%	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
95%	80%	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.99
	85%	0.25	0.25	0.27	0.28	0.27	0.26	0.27	0.27
98%	85%	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00

Table 2. Success Rate of Heuristic on 1Mbp Alignments

		$\mathcal{L}$							
$1-2\epsilon$	$\mathcal{M}$	15	20	25	30	35	40	45	50
70%	55%	6.4	7.9	9.5	11.1	12.8	14.3	15.9	17.5
	60%	6.4	7.9	9.5	11.1	12.8	14.3	15.9	17.5
	65%	6.4	7.9	9.5	11.1	12.8	14.3	15.9	17.5
80%	55%	4.4	5.5	6.5	7.5	8.6	9.6	10.7	11.7
	60%	4.4	5.5	6.5	7.5	8.6	9.6	10.7	11.7
	65%	4.4	5.5	6.5	7.5	8.6	9.6	10.7	11.7
	70%	4.4	5.5	6.5	7.5	8.6	9.6	10.7	11.7
90%	70%	2.7	3.2	3.7	4.2	4.7	5.2	5.7	6.2
	75%	2.7	3.2	3.7	4.2	4.7	5.2	5.7	6.2
	80%	2.7	3.2	3.7	4.2	4.7	5.2	5.7	6.2
95%	80%	1.8	2.1	2.3	2.6	2.8	3.1	3.3	3.6
	85%	1.8	2.1	2.3	2.6	2.8	3.1	3.3	3.6
98%	85%	1.3	1.4	1.5	1.6	1.7	1.8	1.9	2.0

Table 3. Average Wave Span While Finding An Alignment

So we then investigated how quickly the wave's die off after the end of a local alignment with trials where two sequences completely random with respect to each other were generated and then the wave algorithm was called with seed point $(0, 0)$. We recorded the number of waves traversed in each trial, the average span of the waves, and the total number of furthest reaching (f.r.) points computed all together before the algorithm quit. The results are presented in Table 4. Basically the total time to terminate grows quadratically in $\mathcal{M}$ for large values but as $\mathcal{M}$ moves towards the rate at which two random DNA sequences will align (i.e. 48%) the growth in time begins to become exponential going to infinity at 48%. One can begin to see this at $\mathcal{M}$= 55% in the table.

We timed the local alignment algorithm on 15 operating points in Tables 2 and 3 for which the success rate was 100% so that each measurement involved exactly 1billion aligned base pairs. The points covered ϵ from 1% to 15% and $\mathcal{L}$ from 20 to 50. The structure of the algorithm implies that the time it takes should be a linear function of (a) the number of waves, D, (b) the number of f.r. points computed, $D\bar{W}$ where $\bar{W}$ is the average span of a wave, and (c) the number of non-random aligned bases followed in snakes, a. But $D = \epsilon^\star N$ and we know that $i + d + 2s + 2a = 2N$ where i, d, and s are the number of insertions, deletions, and substitutions in the alignment found. The later implies $a = N(1 - (1 + \sigma)/2\epsilon^\star)$ were σ is the relative

$\mathcal{M}$	Average Number of Waves	Average Wave Span	Average Number of F.R. Points
55%	38	24	910
60%	26	24	620
65%	23	22	510
70%	20	20	400
75%	17	17	290
80%	14	14	200
85%	11	11	120

Table 4. Termination Efficiency

portion of the alignment that is substitutions versus indels. Thus it follows that the time for the algorithm should be the linear function:

$$N(\alpha + \beta \cdot \epsilon^\star + \gamma \cdot \epsilon^\star \bar{W}) \tag{3}$$

for some choice of α, β, and γ. A linear regression on our 15 timing values gave a correlation of .9995 with the fit:

$$N(61 + 185\epsilon^\star + 32\epsilon^\star \bar{W}) \text{ nano seconds} \tag{4}$$

For example, with $\mathcal{L} = 30$, the algorithm takes 194s for $\epsilon = 15\%$, 134s for $\epsilon = 10\%$, 91s for $\epsilon = 5\%$, 75s for $\epsilon = 2.5\%$, and 66s for $\epsilon = 1\%$.

To time and estimate the sensitivity of the filtration algorithm we generated 40X coverage of an 10Mbp synthetic genome. Every read was of length 10Kbp and perturbed by $\epsilon = 15\%$ and we sought overlaps of 1Kbp or longer. In Table 5 we present a number of statistics and timings for a few operating points around our preferred choice of $(14, 35, 6)$ for the parameters k, h, and s. The table reveals that (a) the algorithm is very sensitive missing 1 in 5000 overlaps at the standard operating point, (b) the false discovery rate is generally low but does not have a large effect on the time taken by the filtration step, (c) the major determiner of time taken is k, and (d) the time for the filter, e.g. 132 seconds, is small compared to the time taken to find local alignments which was roughly 860 seconds.

(k, h, s)	Sensitivity (TP/(FN+TP)	False Discovery (FP/(TP+FP)	Filter Time (sec.)	Memory (Gb)	Total Time (sec.)
⋆ (14,35,6)	.020%	7.02%	132	11.92	995
(14,32,6)	.014%	7.50%	132		1007
(14,30,6)	.010%	9.51%	132		1014
(14,28,6)	.006%	22.30%	139		1039
(14,35,5)	.037%	6.87%	132	11.92	994
⋆ (14,35,6)	.020%	7.02%			995
(14,35,7)	.015%	7.23%			996
(14,35,8)	.013%	7.84%			998
(13,35,6)	.004%	10.53%	341	12.85	1193
⋆ (14,35,6)	.020%	7.02%	132	11.92	995
(15,35,6)	.109%	6.23%	90	8.22	933

Table 5. `DALIGN` performance on a synthetic 40X dataset as a function of k, h, and s

For all the runs in Table 5, the speedup with 4 threads was 3.88 on average, implying for example that the wall clock time for the standard operating point was 256 seconds, or 4.25 minutes for comparing

two 400Mb blocks. The 40X synthetic data set constituted a single 400Mbp block in the trials, and when compared against itself produced 1.23 million overlaps between the 37,000 reads in the data set. One should note carefully, that for much bigger projects, the time for alignment is considerably less. For example, a 40X dataset over a 1Gbp synthetic genome, would produce 100 400Mb blocks, but comparing each block against itself would typically find only 12.3 thousand overlaps. Another way to look at it is that there will be 100 times more overlaps found, but the filter has to be run on roughly 5000 block pairs.

Real genomes are highly repetitive, implying that the number of overlaps found in practical situations is much higher. For example for the 218Mbp, 31,700 read *E. coli* data set produced by PacBio found 1.44 million overlaps in 1256 total seconds (5.36 wall clock minutes). Moreover, to obtain this result overly frequent k-mers had to be suppressed and low-complexity intervals of reads had to be soft masked. So while the synthetic results above characterize performance in a well understood situation, performance on real data is harder to predict. As our last result, we show in Table 6 the results of timing `BLASR` and `DALIGN` on blocks of various sizes from the PacBio human data set. `DALIGN` was run with $(k, h, s) = (14, 35, 6)$ and k-mers occurring more than 20 times were suppressed. `BLASR` was run with the parameters used by the PacBio team for their human genome assembly (private communication, J. Chin) which were "`−nCandidates 24 −minMatch 14 −maxLCPLength 15 −bestn 12 −minPctIdentity 70.0 −maxScore 1000 −nproc 4 noSplitSubreads`". Reads in the block were mapped to the human genome reference in order to obtain the sensitivity numbers. It is clear that `DALIGN` is much more sensitive (despite the k-mer suppression) and 22 to 39 times faster depending on the block size. In the introduction we gave our time, 15,600 core hours, for overlapping the 54X PacBio human genome dataset, which has been informally reported as 404,000 core hours on the Google "Exacycle" platform using `BLASR` with the parameters as above except `−bestn 1` and `−minPctIdentity 75.0`. This represents a substantial 25X reduction in compute time and returns the problem to a manageable scale.

	BLASR		DALIGN	
Block Size	Sensitivity	Time (sec.)	Sensitivity	Time (sec.)
100	87%	2463	98.7%	109
200	86%	5678	97.5%	222
400	85%	15334	97.3%	393

Table 6. `DALIGN` versus `BLASR`

7 Acknowledgments

I would like to acknowledge Sigfried Schloissnig, who is my partner in building a new assembler for long read data. Also Sigfried's postdoc Martin Pippel produced the timing numbers for the big runs on Arabidopsis and Human, and his Ph.D. student Philip Kämpher produced the statistics for Table 6.

References

1. Myers,E.W. (1986) An O(ND) difference algorithm and its variations. *Algorithmica*, **1**, 251-266.
2. Eid, R., Fehr, A., ... (51 authors) ... Korlach, J, and Turner, S.W. Real-Time DNA Sequencing from Single Polymerase Molecules. *Science*, **323**(5910), 133-138.
3. Lander, E.S. and Waterman, M.S. (1988) Genomic mapping by fingerprinting random clones: a mathematical analysis. *Genomics*, **2**(3), 231-239.
4. Churchill,G.A., and Waterman,W.S. (1992) The accuracy of DNA sequences: estimating sequence quality. *Genomics*, **14**(1), 89-98.
5. Chin, C.S., Alexander, D.H., Marks, P., Klammer, A.A., Drake, J., Heiner, C., Clum, A., Copeland, A., Huddleston, J., Eichler, E.E., Turner, S.W., and Korlach, J. (2013) Nonhybrid, finished microbial genome assemblies from long-read SMRT sequencing data. *Nature Methods*, **10**, 563-569.

6. Pevzner, P.A., Tang, H., Waterman, M.S. (2001). An Eulerian path approach to DNA fragment assembly. *PNAS* **98**(17), 97489753.
7. Kececioglu,J., and Myers,E.W. (1995) Combinatorial algorithms for DNA sequence assembly. *Algorithmica*, **13**, 7-51.
8. Chaisson, M.J., and Tesler, G. (2012) Mapping single molecule sequencing reads using basic local alignment with successive refinement (BLASR): application and theory. *BMC Bioinformatics* **13**, 238-245.
9. Burrows, M., and Wheeler, D.J. (1994) A block sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation
10. `https` : //`github.com/PacificBiosciences/DevNet/wiki/Datasets`.
11. Manber, U., and Myers, E. (1993) Suffix Arrays: A New Method for On-Line String Searches. *SIAM Journal on Computing* bf 22, 935-948.-
12. Cormen, T.H., Leiserson, C.E., Rivest, R.L., and Stein, C. (2009) Introduction to Algorithms (3rd. edition, MIT Press), 197-204.
13. Yuan, W. `http://projects.csail.mit.edu/wiki/pub/SuperTech/ParallelRadixSort/Fast_Parallel_Radix_Sort_Algorithm.pdf`