Bioinformatics
doi.10.1093/bioinformatics/xxxxxx
Advance Access Publication Date: 15 December 2016
Project Report

OXFORD

Consensus Calling

Exploring new Consensus Calling methods for Pacbio data

Swaminathan Sivaraman [1]*, Sriram Sundar [1], Shyam Sundar Chandrasekaran [1] and Prasanth Sridhar [1]

[1] Department of Computer Science, Stony Brook University, Stony Brook, 11794, USA.

*To whom correspondence should be addressed.

Associate Editor: None

Received on December 15, 2016; revised on December 15, 2016; accepted on December 15, 2016

Abstract

Motivation: Long read sequencing technology is being increasingly used, as they have more advantages over short reads in terms of many biological analyses. However, there is also a higher error rate during sequencing and so, long read sequencing technologies work by sequencing a single long read multiple times and calling a consensus for each read. Current consensus calling methods do not do this perfectly and so, we explore alternate methods of consensus generation here (specifically for Pacbio long reads), using our tool called stonyccs. We have created few ordering heuristics and scoring methods and we show how different heuristics and methods affect the final consensus. We have aligned the consensus strings to a known reference genome and presented the accuracy of different methods.

Results: Among our experiments, we found that the best-performing configurations were among the ones that used ordering heuristics based on the STAR algorithm and its variants and scoring methods like an edge-weight-based-scoring method. However, no method performed better than the current consensus calling method, which greatly improves accuracy by precise filtering. We suggest ways of integrating our methods with these filtering methods.

Availability and Implementation: The tool has been developed in Python, and is available at https://bitbucket.org/shyamsundarc92/cse549-compbio. All experiments performed and their results are also uploaded here.

Contact: {ssivaraman,ssundar,shychandrase,prsridhar}@cs.stonybrook.edu

Project details: This project has been performed as part of the CSE 549 - Computational Biology final course project at Stony Brook University, New York, USA.

1 Introduction

Long read sequencing technologies have a lot of advantages over short read ones in terms of whole genome sequencing and other biological analyses. However, there is a higher error rate during sequencing. Current sequencing technologies account for that by sequencing each section multiple times and calling a consensus for each sequence. A variety of methods have been used to perform this consensus calling through the years, the most popular being the Multiple Sequence Alignment or MSA. For a set of sequences, MSA does a pairwise alignment of two sequences and generates a consensus string, and keeps doing a pairwise alignment of this consensus string with subsequent reads. An issue here is that the order of reads which are given to the MSA algorithm will greatly influence the result. Also, simple pairwise alignments do not take into account alignments between multiple sequences.

A new Partial Order Alignment algorithm (Lee *et al*., 2002) was proposed, which created a directed acyclic graph using all the sequences, aligning each sequence to the graph one by one. This POA algorithm and the PO-MSA data structure (the graph) preserved details about multiple sequences and also took into account alignments between any two pair of sequences. Further research work also found ways to generate a consensus string from the PO-MSA graph (Lee, 2003) and ways to order the input sequences for best results (Grasso and Lee, 2004).

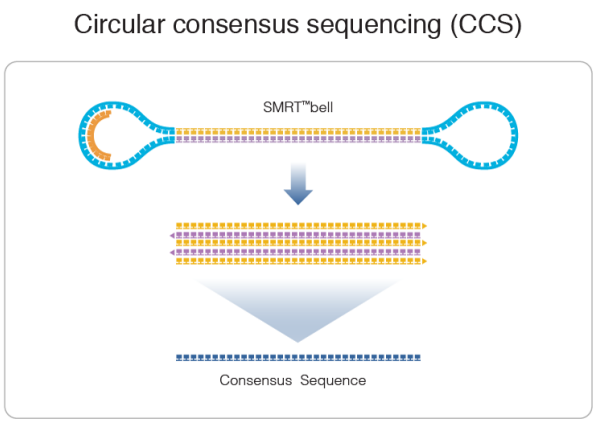

Fig. 1. Pacbio SMRTBell sequencing procedure.

The simple POA algorithm can be used for aligning long reads as well. But the output consensus is greatly dependent upon the order in which the sequences are given to it, and the scoring function and traversal algorithm used to generate the consensus string from the PO-MSA graph. Pacbio provides a consensus calling tool, which uses POA to do the alignment but does not use any ordering heuristics. In this project, we try out different ordering heuristics and also some scoring functions and traversal algorithms different from the ones used by Pacbio, to see if we can generate a better consensus. Most notably, we have used the STAR and a modified version of the STAR algorithm as a ordering heuristic. We have also tried out 2 different scoring functions and 5 different traversal algorithms.

2 Input data

In this project, we are concerned only with the long reads sequenced by Pacbio, and how they sequence the reads and store that information.

2.1 How long-reads are sequenced

Pacbio uses a circular sequencing technique with a SMRTbell Template structure. Here, adapters are attached the forward and reverse strands at each end of the sequence to form a bell shape as shown in Fig. 1. The sequencer just cycles through this bell from adapter to adapter and gives the reads. Note that this means every alternate read will be reverse-complemented.

2.2 How long-reads are stored

Pacbio stores the reads in a bam file, a known format to store reads in a compressed fashion. Pacbio puts most of the metadata about the read like read quality, etc. in the 'tags' field. There is some Pacbio-specific header information also present. During sequencing, for each sequence, Pacbio assigns an id or "well" to it and sequences it multiple times and files all these reads under this well. A ccs tool should call a consensus string for each well from a bam file. Particular fields to note for each field are the read quality, SNR for each channel and the flag information.

2.3 Issues with the input data

We have used the Pacbio data provided for *Arabidopsis*. One major issue we found was that a lot of wells had only one read. This meant that we couldn't do any consensus generation with it, since it is just one string. So, the ccs tool would end up throwing out a lot of data. A second issue was that a lot of reads were not end-to-end reads. Finally and most critically, there was no information as to whether a strand was a forward one or a reverse-complemented one. We could assume that every other read is reverse-complemented but this is not a comfortable assumption to make, and might not be the case, and would completely affect our downstream analysis.

3 Proposed heuristics

3.1 Ordering the sequences

Given the problem and the state of the input data, we devised a set of heuristics. We first needed to use a good ordering algorithm. Given a set of sequences, it must give the best order in which the sequences must be aligned to generate a good PO-MSA graph. General options are to use a STAR algorithm or a guide tree (Grasso and Lee, 2004). But since we don't know about the orientation of the strands, we need to try out more. The proposed algorithms are presented below. We will name each algorithm to aid in easier presentation of test results later on,

3.1.1 Normal STAR algorithm ordering

This is just the normal STAR algorithm. This algorithm assumes that all sequences are forward-oriented. Then it does a pairwise alignment of all sequence pairs and stores the scores. It finds the sequence for which the sum of alignment scores with all other sequences is the maximum (our alignment uses a maximization approach). This sequence is chosen as the first sequence and the subsequent sequences are ordered relative to their alignment score with the first sequence. Henceforth, this algorithm will be refereed to as the *star_fwd* algorithm.

3.1.2 Forward-Reverse STAR algorithm ordering

This is a modification of the STAR algorithm to account for a strand being in either the forward or reverse orientation. In this case, we take the input set of sequences and call it S_f. Then we reverse-complement all sequences and create a new set S_r. Then do pairwise scoring for all elements in both S_f and S_r. That is, for a sequence s_i in S_f, do a pairwise alignment with every other element in S_f and with every sequence in S_r except for s_i's own reverse-complement strand in S_r. We do the same for all elements in S_r as well. Then we find the sequence s_c from either S_f or S_r for which the sum of max(alignment score of s_c with forward-s_i, alignment score of s_c with reverse-s_i), for i going through all sequences but c, is the maximum. That is, choose s_c to maximize,

$$\sum_{i \neq c} max(align\_score(s_c, fwd(s_i)), align\_score(s_c, rev(s_i)))$$

Then after choosing s_c, choose the best orientation of each strand with regards to how best it aligns to s_c. Order the strands using the scores and return s_c followed by the ordered strands. Henceforth, this algorithm will be refereed to as the *star_fwd_rv* algorithm.

3.1.3 Normal Guide tree

This uses a guide tree as suggested in the initial paper (Grasso and Lee, 2004). This algorithm assumes all strands are forward-oriented. It will be called *progressive* (Note that this algorithm will change the way we do PO-MSA itself. Instead of adding a single PO-MSA, we will construct multiple PO-MSAs and combine them in the order instructed by the guide tree. An external library has been used to do this alignment).

3.1.4 Alternate Reverse-complemented Guide tree

This is same as the normal guide tree, but we just reverse-complement every alternate (or odd-numbered) strand in the input sequence well before starting the alignment process. This will be called *alt_rev_progressive*.

3.1.5 Normal Iterative

This doesn't apply any heuristics at all, and just returns the sequences in the order that it gets. This will be called *iterative*.

3.2 Scoring functions

Once the POA graph has been generated, to call a consensus string, we assign a score to each node and also choose the best incoming node for each node. Then, we backtrack from the a 'best possible node' in the graph and return a consensus. The 'best possible node' will be chosen by the traversal algorithm while the score-assignment will be done using the scoring function. The scoring functions we devised are presented below,

3.2.1 Edge-weight-based Score

In this scoring function which will be called *edg_wt*, the score of a node is calculated using these steps:

1. Identifying "node": finding the best predecessor node, i.e. the source node of an incoming edge which has maximum weight (weight of an edge is calculated as the total number of reads passing through both the endpoint nodes connected by the edge)
2. Extracting "Score(node)": once the best predecessor node is identified, extract its score. Note that we are assured to have the score of a node's predecessor computed before computing the score of the current node and all of its successors because we traverse the POA graph in a topologically sorted order.
3. Now, score of "current node" is computed as the sum of the score of its "best predecessor node" and the edge weight between these two nodes as follows,
   Score(node) = Score(best_predecessor_node) + Edge_weight(best_predecessor_node, node)

3.2.2 Pacbio-like Score

This scoring function (called *pb_like*) is similar to but not exactly the same as the one used in Pacbio's ccs tool. Here,
Score(node) = 2*#contained_reads - #outgoing_edges
Note that '#contained_reads' is the total number of reads that pass through the given node. Also, '#outgoing_edges' is different from the concept of 'span' used by PacBio, which they define as the number of reads that pass through a node + number of other reads which pass through at least one ancestor and one descendant.

3.3 Traversal algorithms

This section describes the various node selection algorithms used to determine the starting node from which the backtracking traversal starts building the consensus sequence.

3.3.1 Max Score Node

Start the backtracking traversal from the node with maximum score to generate the consensus sequence. If more than one such node exists, chose the one with max index number (as the nodes are topologically sorted and the last such node would have the max index value). (Algorithm name - *max_sc* or *max_score*)

3.3.2 Max Incoming Edge

Instead of backtracking from the node with maximum score, a node with maximum incoming edge is chosen for backtracking and building the consensus. Nodes with maximum incoming edges are more likely to be the most important nodes in the POA graph as most reads would pass through them. Thus, we can backtrack from one such node. If more than one such node exists, chose the one with max index number (as the nodes are topologically sorted and the last such node would have the max index value). (Algorithm name - *max_edge* or *max_in_edges*)

3.3.3 Random Heuristic based Node Selection

Identify the node with max score (max_score). Fix an optimal value of range (w.r.t max_score) Find a subset of nodes such that (max_score - range) <= score(node) <= max_score (i.e.) find all nodes having scores within a given range of max_score. Randomly select a node from the subset and backtrack. (Algorithm name - *random*)

3.3.4 Max Sequence based Node Selection

Identify the node containing maximum number of sequences and backtrack from that node to build the consensus. If more than one such node exists, chose the one with max index number (as the nodes are topologically sorted and the last such node would have the max index value). (Algorithm name - *max_seq*)

3.3.5 Optimized Random Heuristic

Find a subset of nodes within a specific score range as in method 3.3.3. From the subset, choose a node with max number of incoming edges as in method 3.3.2. (Algorithm name - *opt_random*)

4 CCS Tools

4.1 Pacbio's CCS Tool

Pacbio's CCS tool (which we will call *pbccs*) is written in C++. It accepts a reads file(bam format) as input and calls consensus sequences for each well. This ccs tool does not use any ordering heuristic but uses the following scoring function - Score = (2 * numReads[v] - coverage[v] - epsilon), where epsilon = 0.0001, numReads[v] (or) containingReads = number of reads going through node 'v', coverage[v] (or) SpanningReads = number of reads that pass through 'v' + number of other reads which pass through at least one ancestor and one descendant of 'v'. PacBio uses a "tagSpan" approach to calculate coverage of a vertex, (i.e.) whenever a read is added, the spanning coverage of all vertices "covered" by the read are incremented. Crucially, pbccs does a lot of read filtering based on read quality, SNR, length etc. and uses only a subset of reads to improve the accuracy.

4.2 Our CCS Tool - Stonyccs

We have developed a CCS tool (called *stonyccs*) in Python. It follows mostly the same semantics as pbccs, but it allows us to have an ordering heuristic and also use a different ordering heuristic or scoring function or traversal algorithm if necessary (through command-line options). Stonyccs also performs input read filtering similar to pbccs and the filter parameters can also be modified. Unlike pbccs, stonyccs requires a scoring matrix file like blosum62 or blosum80 (Both matrices are included in the repository). Also, stonyccs requires the input reads fil to be sorted by *queryname*.

5 Experimental results

The reads file provided by Pacbio is around 9 GB in size and it was not feasible to run extensive tests on the full data. So, we extracted reads (total 6 reads) from a particular well and created a sample reads file on which we could run detailed tests for many configurations. We then chose the best-performing configurations and run tests on the full data.

Table 1. A table of blasr scores for alignment of the consensus strings generated by pbccs with –noPolish flag (baseline) and stonyccs to the reference genome for the sample data for different configurations

ordering_algorithm	scoring_func	traversal_algorithm	nMatches	nMisMatches	nIns	nDel	%sim
Pacbio ccs with –noPolish flag (baseline)			1564	63	114	56	87.03
star_fwd	edg_wt	max_score	X	X	X	X	X
		max_in_edges	757	73	86	112	73.63
		max_seq	757	73	86	112	73.63
		random	X	X	X	X	X
		opt_random	X	X	X	X	X
	pb_like	max_score	520	50	31	76	76.8
		max_in_edges	520	50	31	76	76.8
		max_seq	256	21	19	34	77.57
		random	520	50	31	76	76.8
		opt_random	520	50	31	76	76.8
star_fwd_rv	edg_wt	max_score	1922	152	166	178	79.48
		max_in_edges	1848	144	158	162	79.93
		max_seq	1863	147	161	161	79.88
		random	1913	152	164	181	79.37
		opt_random	1892	150	162	175	79.52
	pb_like	max_score	1752	141	122	152	80.4
		max_in_edges	1806	145	122	203	79.34
		max_seq	1752	141	122	152	80.84
		random	879	88	53	70	80.64
		opt_random	1752	141	122	152	80.84
iterative	edg_wt	max_score	1284	133	148	117	76.33
		max_in_edges	X	X	X	X	X
		max_seq	X	X	X	X	X
		random	171	20	19	16	75.66
		opt_random	171	20	19	16	75.66
	pb_like	max_score	62	3	5	11	76.54
		max_in_edges	62	3	5	11	76.54
		max_seq	62	3	5	11	76.54
		random	292	28	32	20	78.49
		opt_random	62	3	5	11	76.54
progressive	edg_wt	max_score	1284	133	148	117	76.33
		random	1264	132	146	116	76.24
		opt_random	171	20	19	16	75.66
	pb_like	max_score	62	3	5	11	76.54
		max_in_edges	62	3	5	11	76.54
		random	292	28	32	20	78.49
		opt_random	62	3	5	11	76.54
alt_rev_progressive	edg_wt	max_score	1742	197	236	313	70.02
		max_in_edges	1095	112	139	180	71.75
		max_seq	1742	197	236	313	70.02
		random	1718	197	232	308	69.98
		opt_random	1742	197	236	313	70.02
	pb_like	max_seq	542	24	39	21	86.58
		random	542	24	39	21	86.58
		opt_random	542	24	39	21	86.58

'X' denotes no match. All tests have been run using the blosum62 scoring matrix

5.1 How results are measured

For each consensus pbccs or stonyccs generates, we map it to the reference genome (location specified in the REASME file) using Pacbio's *blasr* tool. This reports the number of matches, mismatches, insertions, deletions and a percentage similarity score.

5.2 Sample data test results

We extracted a set of 6 reads from the full data and constructed a sample reads file (can be found at *sample_tests/sample_bam.bam* in the repository). We then ran various tests for different configurations.

5.2.1 Tests with filtering disabled

Now, pbccs filters out most of the input reads. We hypothesized that we would get a better consensus if we used more data. So, we ran stonyccs with 6 configurations (results are in the *sample_tests/all_filters directory*) after disabling all input read filters. But the tests returned an average similarity percentage of 75%. On running the tests with input filters enabled (present in the next section), we found that we could get better scores with lesser data. This is probably because the reads that are filtered out are typically low-quality reads and will mostly not serve to better the consensus, but will definitely slow down the run-time of the program. As algorithms like STAR and the guide tree take a long time for a large number of sequences,

Table 2. A table of blasr scores for alignment of the consensus strings generated by pbccs with –noPolish flag (baseline) and stonyccs to the reference genome for the full data for 12 chosen configurations

ordering_algo	scoring_func	traversal_algo	Total wells	Avg. nMatches	Avg. nMisMatches	Avg. nIns	Avg. nDel	Avg. %sim
Pacbio ccs with –noPolish flag (baseline)			63	1999	94	102	104	85.813
star_fwd	edg_wt	max_seq	61	1821	140	162	159	80.706
	pb_like	max_in_edges	61	1625	121	115	153	81.251
star_fwd_rv	edg_wt	max_score	61	1839	117	145	145	82.416
		max_in_edges	61	1683	110	125	133	81.977
		opt_random	61	1822	121	146	147	82.504
	pb_like	max_in_edges	61	1667	104	101	134	82.201
		max_seq	61	1669	105	99	137	82.245
		random	61	1140	76	75	87	82.713
iterative	pb_like	max_seq	61	1940	155	195	180	79.696
progressive	pb_like	random	61	974	68	76	89	81.733
alt_rev_progressive	edg_wt	max_in_edges	61	1754	133	155	181	79.971
	pb_like	max_seq	61	1633	120	111	190	80.535

All tests have been run using the blosum62 scoring matrix

we decided to always have the input filters and modify only the ordering heuristic, the scoring function and the traversal algorithm.

5.2.2 Tests with filtering enabled

Stonyccs performs the following input read filters for each well,

- Check if there are more than 3 reads
- Only allow end-to-end reads
- Only allow reads with read quality, $rq \geq 0.8$
- Only allow reads with SNR ≥ 3.75
- Only reads where $10 \leq read\_length \leq 7000$
- Drop reads with length beyond a certain percentage of the median of all read lengths.

There are some more pbccs filters like Z-Score and PredictedAccuracy filters which stonyccs does not implement.

We ran stonyccs on the sample bam file for all ordering_heuristic-scoring_function-traversal_algorithm combos. We ran the tests for both blosum62 and blosum80 matrices. We observed that the blosum80 results generally weren't being matched at all, and when they were, the nMatches was very low though %sim was high. So, we ran thorough tests only for the blosum62 matrix. All results using the blosum62 matrix are presented in Table 1.

None of the configuration's similarity scores exceeded that of pbccs's (*baseline*) scores. However, some of the configurations are coming quite close and some configurations are also matching more characters than the pbccs version. A simple scan of the table tells us that the *star_fwd_rv* is the best-performing ordering heuristic. It gives a high similarity score + high nMatches for most of the cases. The *alt_rev_progressive* heuristic gives the best similaraity score but it's nMatches is almost one-fourth of *star_fwd_fv*. The *iterative* heuristic performs the worst. With regards to the scoring functions, the *edg_wt* function is performing marginally better than the *pb_like* function. The *max_sc* and *max_seq* traversal algorithms perform better for some cases while the *max_in_edges* and *opt_random* performs better for some other cases. We chose 12 best configurations from among these, making sure to also cover as many different algorithms as possible without giving up on the scores. We ran these 12 configurations on the full data set (All sample data test results are in the *sample_tests* directory).

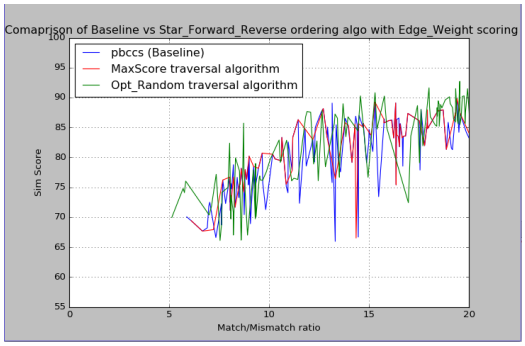

Fig. 2. A plot of matches/mismatches-sim_score for pbcss and 2 best stonyccs configurations

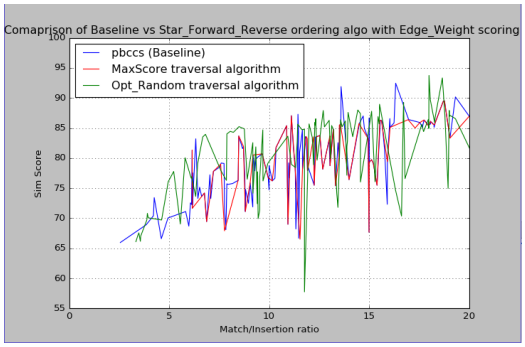

Fig. 3. A plot of matches/insertions-sim_score for pbcss and 2 best stonyccs configurations

5.3 Full data test results

We ran the tests on the full data for these 12 configurations on a subreads.bam file from Pacbio (Specifically, the sorted version of this bam file. The exact location of this data is specified in the README file). Stonyccs's input filters chose 61 wells to run CCS on and ran CCS for each well. We then generated blasr reports for each case and ran a simple script to read the report and average the 61 independent results for each case. All of these have been run using the blosum62 matrix and the results have been tabulated in Table 2.

As seen, now our results are somewhat close to the baseline, though none beats the baseline. Some results have lesser number of insertions but the number of matches also decrease for that case. The best ordering heuristic is the *star_fwd_rv* algorithm. Based on the number of matches and the similarity score, the best-performing scoring function is the *edg_wt* scoring function and the two best traversal algorithms are *max_score* and *opt_random*. Though the number of insertions is generally high for stonyccs's configurations. To see how the number of mismatches and insertions vary with respect to the matches count, we plotted matches/mismatches-sim_score and matches/insertions-sim_score in Fig. 2 and Fig. 3. It can be seen that no one algorithm is completely solid. For high match/mismatch values, the stonyccs algorithms are performing better than the pbccs version. For the match/insertion values, there is no solid winner and all algorithms are fluctuating. But for high match/mismatch values, the *max_score* is more stable than the other two algorithms.

6 Discussion

Our algorithms seem to perform better when we have more number of matches, that is, we seem to have higher accuracy for longer consensus strings.

6.1 Possible reason for not being able to beat the baseline

Stonyccs' average case scores are less than that of the pbccs version. This could possibly be because of the Z-score (a Pacbio terminology) filtering that pbccs does. It does this after the consensus and removes reads that have a lesser Z-score and re-runs the consensus. Possibly adding this Z-score filtering step to our algorithms can improve the result.

6.2 Performance of stonyccs

Stonyccs runs in around 5-6 minutes for the normal iterative algorithm cases. When run with the different STAR or progressive algorithms, it takes around 14-15 minutes. This is because it has to do pairwise matching for every sequence pair and every reverse-complemented pair also for the *star_fwd_rv* case. As this is real data however, 15 minutes looks to be acceptable.

6.3 Future work and suggestions

We wanted stonyccs to use more reads than what pbccs uses to do the consensus. But stonyccs ended up using lesser reads than the number of reads pbccs used. So, we may need to see where we can relax stonyccs' filters. Also, for wells with lots of reads, the STAR algorithm will be quite slow and efforts can be made to speed this up.

7 Conclusion

We have explored different ordering heuristics, scoring functions and traversal algorithms to improve the consensus calling of Pacbio long reads. We were able to achieve results that were close to the baseline, but couldn't beat the baseline. Future exploration can be done to see if incorporating pbccs' filters properly into our tool can improve the results.

Acknowledgements

We thank Avi Srivastava, our adviser, for providing a detailed explanation of how Pacbio performs long-read sequencing. We also thank our professor, Dr. Rob Patro, whose classes inspired us to take up this project. Finally, we thank Pacbio for making both their datasets and consensus calling tools publicly available.

References

Lee,C., Grasso,C. and Sharlow,M. (2002) Multiple sequence alignment using partial order graphs, *Bioinformatics*, **18**, 452-464.

Lee,C. (2003) Generating consensus sequences from partial order multiple sequence alignment graphs, *Bioinformatics*, **19**, 999-1008.

Grasso,C. and Lee,C. (2004) Combining partial order alignment and progressive multiple sequence alignment increases alignment speed and scalability to very large alignment problems, *Bioinformatics*, **20**, 1546-1556.