Preliminary Studies on de novo Assembly with Short Reads

Nanheng Wu
Satish Rao, Ed.
Yun S. Song, Ed.

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2009-172
http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-172.html

December 15, 2009

Acknowledgement

I want to dedicate my thanks to Professor Satish Rao and Professor Yun Song their guidance and mentorship throughout this study. I thank Wei-Chun Kao for valuable input on the clustering algorithm. I thank Matei Zaharia for helping get started on Amazon MapReduce and providing answers to numerous related questions. I also thank Aileen Chen and Jerry Hong for their dedication on implementing solutions and conducting experiments.

Preliminary Studies on de novo Assembly with Short Reads

Abstract

Recent development of next generation sequencing presents new computational challenges to assembly algorithms. Any effective and practical de novo assembly algorithm must confront issues of short read length, base-calling errors and enormous data size. In this report we present our effort to address these challenges in de novo assembly with short reads. Specifically we show that quality scores contain vital information and algorithms can achieve optimized results if they utilize quality scores. We also show that error correction preprocessing can be used to enhance de novo assembly algorithms with more tolerance to base-calling errors. Finally we present a novel parallel algorithm to cluster sequence reads based on overlap information and show that it has the potential to scale up to handling millions of reads efficiently.

1. Introduction

New emerging DNA sequencing technologies can produce short length reads at an unprecedented level of throughput than traditional methods. For instance, massively parallel sequencing platforms produced by Illumina and Roche have delivered sequencing capability of up to several Giga base-pairs of DNA per single run. Next generation sequencing technology brings new challenges to the task of analyzing its results. First of all, the output sequence reads are typically much shorter than those produced by the traditional Sanger technology. Shorter reads naturally carry less information. Second, high level of throughput produces enormous number of output sequence reads. Typically, the number of sequences generated is on the order of tens of millions. Therefore, any program that processes such data must be highly efficient and scalable.

DNA sequence assembly with short reads aims to use the short sequence reads to reconstruct the genome that is being sequenced. The goal of de novo assembly is to perform DNA sequence assembly without comparing the reads to a reference genome. De novo assembly with short reads has been an active area of research, and many algorithms have become available recently. Besides the typical difficulties such as short read length and large input dataset size de novo assembly with short reads faces another challenge: base-calling errors. Errors often occur in the process of extracting sequence information from the underlying signal produced by the sequencing platform. Such errors drastically increase the complexity of many assembly algorithms and cause them to perform worse with increased numbers of mis-assembled contigs or decreased number of bases from the original genome that are sequenced.

In this study we investigate several recently published de novo assembly algorithms and analyze the flaw in these existing algorithms. One weakness shared by many existing algorithms is their low tolerance to base-calling errors and another is their lack of use of the quality information in the data. We attempt to help improve the existing algorithms and help them contend with the computational challenges in three areas: First, we show that although short reads carry less information in the sequence by nature, quality scores produced by the sequencing platforms can provide vital information. In particular we show that we can improve the performance of widely used software VELVET by making it utilize the quality information in sequence reads. We modify portions of VELVET's error correction algorithm to compare sequences based on the likelihood of each base and observe a significant boost in VELVET's accuracy. Although most current algorithms make an effort to handle base-calling errors, such errors remain a challenge to de novo assembly algorithms with short reads. Therefore in our second area of focus, we show that an independent error-correction preprocessing stage of the data can help algorithms tolerate base-calling errors. We develop an error correction method using graph theoretical technique and demonstrate improvement on the performance of VELVET in both accuracy and percentage of bases assembled. The impact of our method is particularly apparent in the regions where the error rate is high; in such region VELVET can barely produce any assembled sequence before error correction and gives reasonable results after we clean the data with our method. Last, we turn our attention to developing a reliable and scalable de novo clustering method with short reads. The goal of the method is to aggregate reads that are from the same region of the genome in clusters. The clusters produced by our methods can be used by statistical methods to perform error correction on short reads. We implement a parallel algorithm that first bins reads into approximate clusters and uses graph based methods to refine them into actual clusters. The method is implemented with Hadoop and we test it running on Amazon's cloud computing cluster. The method demonstrates excellent clustering results and high scalability.

2. Related Works

The area of de novo assembly with short reads has received much interest recently and many new algorithms have been proposed to solve the problem. Our primary focus in the study is graph theory based algorithms because they generally perform better. Moreover, graph theoretical techniques are well studied and understood. The approaches taken by current algorithms fall into two general categories: *read* graph or *de Bruijn* graph. In a read graph (or sometimes referred to as *overlap graph*) based model each input read is represented by a node in the graph and two nodes are connected by an edge if the reads overlap by at least a minimally allowed number of bases. EDENA (Hernandez et al, 2008) is an algorithm that explicitly uses this framework and other assemblers such as SSAKE (Warren et al, 2007) use this framework implicitly.

There are a number of issues that make the assembly process challenging using read graph based approach. As pointed out in Butler et al. read graphs depend on the choice of minimum overlap size (K). If K is chosen to be too small there are too many "accidental" overlaps, meaning that two reads overlap by K bases but they do not really belong in the same region of the genome. The implication of choosing a small K is that the graph becomes very dense and it is computationally difficult to distinguish the "true" overlaps so that the algorithm does not glue along the wrong reads and produce incorrect contigs. On the other hand if K is chosen to be too large, although it drastically increases the reliability of an overlap, such overlaps become increasingly rare as K gets larger, therefore without ample coverage the graph becomes sparse and makes it a difficult task to assemble large contigs which are useful in real biological applications.

Base-calling errors further add to the complexity of solving the short read assembly problem using read graph based approach. This is because erroneous reads create random branches that may connect regions that are supposed to be disjoint (with no overlapped reads) and mis-assembly can happen if the errors are not identified and handled carefully.

To correct sequencing errors some read graph based algorithms take the approach of cutting off branches in the graph that dead end quickly. EDENA is one with such approach. The basis of this approach is the assumption that erroneous reads are less likely to have overlapping error-free reads and therefore paths that only extend to a small number of nodes are likely to contain reads with errors thus removing such paths eliminates the errors. However, there has been no published work on how effectively this technique works and therefore it is hard to determine under what conditions this technique works beyond an intuitive level. SSAKE does not deal with errors directly, the algorithm greedily extends the contig with overlapping reads; whenever ambiguity arises at a position the algorithm uses consensus voting from all the reads that cover the position to decide on the correct base. The consensus voting scheme can be a mechanism to deal with errors by taking advantage of the coverage depth. The problem this approach suffers from is that it uses very little information about each read such as how many *distinct reads* overlap with it: if a region of the genome is repeated or approximately repeated several times SSAKE's approach quickly leads to the wrong result once it includes reads that do not belong in the same region.

De Bruijn graphs take a different approach to represent the input sequences. In a de Bruijn graph model the representation of the input data is not organized directly around the reads but around substrings of reads of length *k*, or *k-mers.* Each input sequence is mapped to a path that "threads" through all if its *k-mers.* The de Bruijn graph data structure is attractive because it is based on *k-mers* not reads and it can encode redundancy in the data without growing the size of the graph. It is natural to represent the multiplicity of a read with the weights on all the edges that connect the read's *k-mers.* As a trivial example, for k=3 a read ACGT can be encoded in a de Bruijn graph with (ACG$\rightarrow$CGT). If the read appears twice in the data, 2 would be the weight of the

edge. The de Bruijn graph structure can also map repeat reads easily: they simply correspond to paths that share some nodes but with different start and end points.

Much work has been done to suggest ways of using the de Bruijn graph structure; EULER-USR and VELVET are the two popular ones among them. These assembler programs both explicitly use the de Bruijn graph framework with notations of dealing with errors. The technique EULER-USR uses to correct read errors transforms the problem into solving the "Spectral Alignment Problem." It is based on the fact that the beginning portion of the reads are much more reliable and almost error-free. Therefore, it extracts from the reads the set of *k-mers* that appear most frequently and deem them as error-free *k-mers,* it next uses the error-free *k-mers* to correct errors in the rest of the *k-mers* by doing minimal amount of substitution or mutation (Chaisson, *et al* 2008). A read is considered corrected if all of its *k-mers* have been corrected and included in the error-free set. We note that a similar error correction approach is used by another algorithm ALLPATHS which does not use the de Bruijn graph structure. ALLPATHS looks at the distribution of *k-mer* multiplicity and looks for the local minimum that separates correct and incorrect *k-mers.* It defines a *k-mer strong* if it has multiplicity greater than such local minimum *m*. The error correction uses a series of values of k and checks for each k if all of those *k-mers* in that read are strong. If they are, the read is considered correct and left alone, otherwise a correction method is used to correct the read by making "one or two substitutions" (Butler et al 2008). If such correction is unsuccessful and the read cannot be corrected by some small number of changes it is discarded before assembly. Lastly, the error correction process in Velvet is more complex and it involves several graph based techniques such as removing tips, bulges and removing "bubbles".

In order to evaluate the performance of each assembler our study uses mainly two metrics: to measure how much of the original genome can be effectively reconstructed by the algorithm we count the number of bases covered by contigs, a contig is a larger string formed by piecing the input reads together and a base is covered by a contig if it appears in it; in order to evaluate the accuracy of each assembler we examine their error rate, which is the number of contigs that contain reads which do not belong together but are merged incorrectly. The results suggest that de Bruijn graph based algorithms generally perform better than their read graph based counterparts on both metrics and EULER-USR have comparable performance to Velvet, although it appears to work better under certain conditions.

We note that although there has been substantial effort made in the development of de novo assembly using short reads, base-calling errors still remain a challenge for current algorithms. Many algorithms today have low tolerance for base-calling errors and they perform drastically worse even with small increase of error rate. We see two reasons that explain this low tolerance of errors: First, although many algorithms acknowledge the importance of error correction their procedures are mostly ad hock, as far as we understand no existing algorithms try to use information from the data to the fullest

degree. Second, the error correction procedures used by existing algorithms are computationally expensive and they are infeasible as the input typically consists of tens of millions of reads. Therefore we concentrate our effort to improve de novo assembly with short-reads algorithms in three areas. First, we show that existing algorithms can benefit if they harvest the quality score information of the data. For this task we modify the appropriate portion of source code of the VELVET assembler to incorporate reads quality into its bubble removal algorithm (named Tourbus). Our results show considerable improvement on the accuracy for VELVET, suggesting that incorporating quality scores in other sections of the algorithm is an area worth exploring. Second, we show that an error correction preprocessing of the data can lead to improvement on existing algorithm without any modification. We approach this task by developing an error correction program based on majority voting and feeding the corrected reads into VELVET assembler. The effect of the error correction procedure is another notable improvement on accuracy and number of covered bases. Last, we focus our effort on developing a reliable and scalable de novo clustering procedure that is suitable to process millions of reads. We develop a parallel algorithm using the MapReduce framework to run on any Hadoop cluster. The result of our clustering algorithm can be directly used by statistical methods to perform error correction on large data sets. The success of our clustering algorithm suggests a new way parallel applications can be applied in the space of DNA sequencing and we believe this is going to inspire the discovery of new research opportunities in this area.

3. Improving de novo assembly with short reads

3.1 Incorporating Quality Score in VELVET

Quality Scores

In Modern Sequencing platforms the produced short reads also include information about the likelihood of each base. For instance, the Illumina platform outputs this information as Quality Scores. For each read produced by the platform there are four quality scores for every position, one score for each base (A, C, T or G). Such base-specific score $Q(x)$ is defined as:

$$Q(x) = 10\log_{10}[\frac{P(x)}{1-P(x)}],$$

where $P(x)$ is the probability that the base is the correct one at the current position. It is easy to derive the probability given a quality score:

$$P(x) = \frac{10^{\frac{Q(x)}{10}}}{1+10^{\frac{Q(x)}{10}}}$$

For instance in Illumina based technology if a position has quality scores {“A”=40, “C”=-40, “G”=-40, “T”=-40} it indicates that the correct base at that position is almost certainly “A”, whereas quality scores {“A”=-4.7, “C”=-4.7, “G”=-4.7, “T”=-4.7} suggest that the correct base could be any of the four bases with equal likelihood. Quality scores in

reads allow us to use the information in the data to the fullest extent, as we show in the next section one of the applications of quality scores is to aggregate the scores for each position that is covered by multiple reads and to infer the most likely outcome in the end. This idea is similar to majority voting but instead of a binary vote each read contributes a probability. We observe a substantial improvement when we apply this technique to the Tourbus algorithm of the VELVET assembler.

Tourbus algorithm

VELVET uses a collection of graph based techniques to correct errors including removal of tips and bulges and resolution of bubbles. A "bubble" in a graph is defined as two paths that share the same end-points but are otherwise disjoint. Recall that VELVET uses the de Bruijn graph approach where each read is mapped to a path that threads together its k-mers. A bubble in the de Bruijn graph may be caused by errors in some reads that create another path, which diverges from the correct one. The Tourbus algorithm is VELVET's technique to identify the erroneous path and merge it with the correct one and remove the bubble from the graph (Zerbino DR, Birney E. 2008). The Tourbus algorithm is essentially a Dijkstra-like breadth-first algorithm. The algorithm cycles through all the nodes and for each node it runs a shortest path algorithm where the distance is defined by the number of nodes on the path divided by the path's multiplicity. If the algorithm reaches a node whose closet neighbor has already been previously visited the algorithm backtracks to find the bubble. It follows the backward edges simultaneously from the current node and the previously visited neighbor until the two "backward" paths converge at a single node. Once the bubble is discovered the algorithm extracts the sequences represented by the two paths and aligns them using a dynamic programming algorithm; if the two sequences are judged similar enough (by comparing the score of the best alignment to a fixed parameter of the algorithm) the Tourbus algorithm goes on to merge the two paths and remove the bubble.

Our modification

The Tourbus algorithm uses a standard dynamic programming algorithm to compute the similarity between two sequences, in which assigns Indels and mis-matches equally with score of 0 and matches with score of 1. We discover there an opportunity for improvement if we incorporate quality scores into the scoring matrix. The intuition is the following: If the two positions have bases with high likelihood the penalty should be higher than when it is less certain what the correct bases are. Suppose the quality scores for the two reads we are comparing are {"A"=40, "C"=-40, "G"=-40, "T"=-40} and {"A"=-40, "C"=-40, "G"=-40, "T"=40} at some location, this means that one base is almost certainly an "A" and the other one almost certainly a "T". This mis-match should be given a penalty higher than scores like {"A"=-40, "C"=0, "G"=0, "T"=-40} and {"A"=-40, "C"=0, "G"=0, "T"=-40} where the correct nucleotide could be C or G with equal uncertainty. It is apparent here that without the use of quality scores the original Tourbus algorithm could not differentiate such situations.

In order to inject quality scores into VELVET we make considerable amount of changes to the graph data structure and the Tourbus algorithm. In the graph data structure we make the following modification:

1.To store quality scores of each base, we create an additional field *scores* in the original structure that stores data of a node *node_st*. The field *scores* is an array of *qualityScore_st* pointers. Each *qualityScore_st* corresponds to a base that is contained in the graph node, each *qualityScore_st* has the 4 scores and a counter to keep track of how may values have been accumulated at the position, it is used for probability accumulation.

2. To aggregate quality scores of each sequence we create a function *accumulateProb* that takes in a new set of scores and add it to a specific position of a graph node. In this experiment we compute a new set of averaged quality scores of all the scores we have seen so far and convert the scores into probabilities. The VELVET function that imports all the reads and constructs the de Bruijn graph is in *threadSequenceThroughGraph* function. In this function the algorithm processes each read and maps it to a path in the graph. We first modify VELVET to accept an additional file of quality scores and we then modify *threadSequenceThroughGraph* function so that when each read sequence is mapped through a path we use the *accumulateProb* function to compute the averaged scores.

3. We need a mechanism to output a string sequence according to the quality scores at each position. We insert code in *exportLongNodeSequence* function to output the most likely base (one with highest quality score) at each position. VELVET uses this method to output the final assembled contigs and our modification allows the algorithm to produce the result according to quality information.

The modification of the Tourbus algorithm is a nontrivial software engineering task. When the algorithm merges the two disjoint paths all the information stored in the node such as edges, multiplicity information and quality scores must be updated with care to ensure the data structure consistency. In our approach we create a new representation of a sequence, called QualityScoreSequence to replace the original character based sequence representation. A QualityScoreSequence is essentially an array of pointers to the quality scores of each base. The Tourbus algorithm starts from a random node in the graph and runs a Dijkstra algorithm from it. If it encounters a node that has been previously visited, it is an indication that it has encountered a “Bubble”. To remove the bubble the Tourbus algorithm traces backwards to find the two paths that share the same end points. Tourbus defines the path with shorter distance as the “fast” path and the other one the “slow” path. It proceeds to extract the sequence from each path and compute their similarity score and alignment with dynamic programming. The original Tourbus extracts a character based sequence from each path, we modify the algorithm

such that it creates a *QualityScoreSequence* from each path. This modification enables us to use quality information in the alignment computation. The original Tourbus computes all possible alignments and assigns scores for each position as follows: if the bases at each position match the score is 1 otherwise it is 0 (mis-matched base and gaps receive the same score). The final alignment is the one with the highest total score. Our approach uses the *QualityScoreSequence* to compute the alignment, in our dynamic programming code we assign scores for each position using the following method: a gap receives a score of 0 as in original Tourbus, otherwise we compute the product of the quality scores of all four corresponding bases at the position and use the sum of products as score. We also use the alignment that maximizes the score as the final alignment. Once the Tourbus algorithm identifies the two sequences and computes the similarity score for them it moves to merge the two paths that the sequences map to. We also modify this portion of the Tourbus algorithm to merge our *QualityScoreSequence* data structures so that we merge the quality scores correctly. When Tourbus transfers information from the "slow" path nodes to corresponding "fast" paths nodes the multiplicity increases for "fast" path nodes and we use the *accumulateProb* procedure to add scores from the "slow" path node to the "fast" path node.

Results and discussion

In our preliminary study we set up an experiment using short reads from a randomly simulated genome. We chose various coverage depth and error profiles. The "flat" error profile has a uniform error rate for all positions of an input sequence and "Bustard" error profile models the realistic sample from Illumina platform which produces most of the errors at the end of each sequence. We also tried "10X Bustard" error profile to simulate pair-end data, we notice when paired-end data is used the first read generally has good accuracy but the second read tends to have a much higher error rate. All the tests in this study are sampled from a simulated genome of 1000 bps and with read length of 76 base pairs.

The results showed that we achieved a significant improvement on accuracy for the VELVET assembler for all 3 error-profiles as a result of incorporating quality scores in the Tourbus algorithm (Table 1).

Table 1: **Comparison of VELVET assembly results using quality scores**

		Bustard Error Profile			10% Flat Error Profile			10X Bustard Error Profile		
Coverage	k	QS	No QS	Imp %	QS	No QS	Imp %	QS	No QS	Imp %
5	21	0.3%	0.5 %	40%	6.5%	8.3%	22%	1.3%	2.4%	46%
5	29	0.2%	0.3 %	33%	1.5%	4.0%	63%	0.3%	1.2%	75%
10	21	0.3%	0.5 %	40%	7.7%	12.3%	37%	1.5%	2.4%	38%
10	29	0.1%	0.3 %	67%	1.7%	3.4%	50%	0.5%	1.1%	55%

The columns labeled "QS" corresponds to the VEVLET algorithm error rate with quality scores incorporated, and "No QS" corresponds to the error rate when quality scores are not used. Imp% denotes percentage of improvement. Every experiment here is repeated 50 times and the average values are reported

We also observed improvement on contig sizes but they were not substantial. Our study on the VELVET software has put us in the position to deliver more in depth modification in other areas of the algorithm and study the full effect of quality scores in future research. For instance, in the Tourbus algorithm the length of a path is calculated using a heuristic that takes into account the number of nodes on the path and the path's multiplicity. When merging two paths the path with a shorter distance is considered "fast" the other one "slow". The Tourbus algorithm destroys the "slow" path by transferring information from the "slow" path to the "fast" one. We think the path selection process can also use the quality information to decide on the more likely path.

3.2 Error correction preprocessing

In this section we discuss the second aim of our research to improve de novo assembly algorithm with short reads by contending with base-calling errors. Base-calling errors occur when the sequencing platform fails to convert the underlying signal to the corresponding base. These errors introduce noise to the data and negatively affect de novo assembly algorithms to produce shorter assembled sequences or more assembly errors. Although some measures are taken to deal with sequencing errors, they still remain a serious challenge in the area of short reads assembly. Some algorithms use an explicit error correction stage to clean up the data before assembly such as ALLPATHS and EULER-USR. In particular, the EULER family algorithms attempt to correct the errors by formulating it as "Spectral Alignment Problem". The solution uses the coverage information by finding k-mers that occur frequently in the data set and correct the other k-mers with a dynamic programming based algorithm [3]. Such algorithms are limited by low coverage and low efficiency and therefore not practical on large data sets. VELVET however does not have an error correction preprocessing; instead it attempts to correct the errors through a series of techniques that are built in as part of the algorithm. We organize a study on the impact of error correction preprocessing on VELVET. In this study we create an error correction preprocessing algorithm and deliver the corrected data to VELVET. Our study shows that the strategy is effective in general yielding fewer errors and more bases from the genome to be assembled into contigs. Note that our method leads to great results especially when the coverage is low (< 10X coverage). This is an attractive property because this allows more genomes to be sequenced at low cost, for instance the "1000 genome" project (1000genome.com) plans to sequence the genome of one thousand individuals at coverage depth of 3-5X, because such coverage depth is below the level at which VELVET can perform well we believe our error-correction method can be of much value in such projects.

Our error correction method

Our method corrects errors in input reads by correcting errors that occur in k-mers. The input to our algorithm is a set of sequences and the desired k-mer length to use where k is a parameter of the algorithm. Our algorithm first applies hashing techniques to find similar (identical except a few bases) k-mers. It proceeds as follows: for each input

sequence the algorithm extracts all possible k-mers, if the input sequence has length L there are $L-k+1$ k-mers per sequence. For each such k-mer the algorithm stores it in a hashtable with the ID of the sequence it comes from and the offset from the beginning of that sequence. Once the hashing phase is complete the algorithm collects all the k-mers from the hashtable and computes the hamming distance between all pairs of k-mers. The intuition is to divide k-mers that are very similar into clusters. The algorithm accomplishes this by constructing a graph in which each k-mer is a node and two k-mers are connected by an edge if their hamming distance is less than a predefined threshold. The all-pair distance computation allows the algorithm to construct all the edges. Once the graph is created our algorithm computes the connected components on the graph and each connected component contains k-mers that only differ by few bases. The next step is to build the prototype k-mer from each cluster; our method uses a consensus voting scheme to effectuate that. We do not use the multiplicity in this implementation so every node contributes one vote. We assume that the prototype k-mer is error free and thus our method uses the prototype k-mer to correct the errors in each sequence. Since we store the reads and starting offset of each k-mer in the hashtable error correction becomes a trivial task. We compare the k-mers of each cluster with its prototype one position at a time, if they differ at any position since we know that the position in the read is simply the position in the k-mer plus the offset of the k-mer in the read, we replace the base of the read with the one from prototype.

Experiments and Preliminary Results

We simulated genomes of various sizes for this experiment and tested our method using simulated sequences of various lengths and depth of coverage. We also applied two different error profiles, “Flat” is a uniformly random error profile and “Bustard” models after the realistic data sample from the Illumina platform. We also coarsely simulate the error profile of paired-end data with “10X Bustard” profile because typically in paired-end data the second read has much higher (usually around 10X) error rate than the first read.

We ran VEVLET on both uncorrected and corrected reads and noticed significant improvement in both the number of bases assembled and accuracy. We observe that in case of relatively low error rate such as the case of Bustard model VELVET fared slightly better in terms of bases covered with corrected data and substantially better in terms of accuracy. Also note that in case of high error rate VELVET does very poorly (0 assembled bases under 10% Flat Error Profile) and error correction brought drastic improvement in number of bases assembled (17X under 10X Bustard Error Profile).
Our results show that VELVET without help of the error correction preprocessing is very sensitive to base calling errors in the reads. As the errors reach a certain threshold none of VELVET’s internal error correction mechanisms are effective.

For instance, in the case of 10% Flat Error Profile (Table 3) too many errors in the data create many dead-ended branches and short cycles which get removed by VELVET and therefore it wasn’t able to produce any output.

Table 2: **Comparison of VELVET assembly results in the case of relatively low error rates corresponding to Bustard error profile**

				Bases Covered			Error Rate		
Genome Size	Read Length	Coverage	k	Corr.	Uncorr.	Imp %	Corr.	Uncorr.	Imp %
500	60	10	21	483	469	3%	0.03%	0.33%	91%
1000	76	5	21	923	881	13%	0.19%	0.41%	54%
1000	76	5	29	823	784	5%	0.27%	0.31%	13%

The column labeled “Corr.” corresponds to VELVET assembly result with corrected reads using our method, “Uncorr.” corresponds to results using uncorrected reads and “Imp%” corresponds to percentage of improvement.

Table 3: **Comparison of VELVET assembly results in the case of high error rates corresponding to 10% uniform error and 10X the Bustard error profile**

				Bases Covered			Error Rate		
Genome Size	Read Length	Coverage	k	Corr.	Uncorr.	Imp %	Corr.	Uncorr.	Imp %
(10% Flat Error Profile)									
500	60	10	21	378	275	37%	2.4%	4%	41%
500	60	10	29	241	0	∞	1.5%	N/A	N/A
1000	76	5	21	844	547	54%	4.1%	8.2%	50%
1000	76	5	29	504	73	591%	2.9%	3.5%	16%
(10X Buster Error Profile)									
500	60	10	29	362	20	1700%	0.52%	0.30%	−73%
1000	76	5	21	645	312	106%	2.2%	2.3 %	4.3%

See Table 2 for explanation of column labels. This table shows that under high error rates the fraction of bases covered by the assembled contigs are much greater after error correction. With exception of only one case accuracy is also improved with error correction; note that increasing fraction of bases covered should be expected to reduce accuracy because the contigs must cover regions with more error.

Note that although sequencing technology has improved on base calling errors in practice it is very difficult for users to consistently achieve the advertised accuracy and high error rates such as 10% is not atypical. Our study demonstrates that error-correction can serve as a valuable preprocessing stage to clean up data with high error rate to allow users to make use of the otherwise unusable data.

The challenge of developing a reliable error correction method is that it must handle millions of sequences efficiently. Our proof-of-concept implementation has running time quadratic to the number of reads that is not feasible in reality. In the next section we discuss our approach to developing a reliable and scalable de novo clustering algorithm. The output of our clustering algorithm can be directly used to perform error correction on large data sets.

3.3 Parallel de novo Clustering with Short Reds

In the previous section we highlight our study on error correction preprocessing and its potential effect on de novo assembly algorithm with short reads. We use a popular

Figure 1 Example of short reads clusters. In this figure is a genome of 23 bps. 15 short reads have been sampled from the genome. Reads that belong in the same circle are from the same region of the genome and therefore are in the same cluster.

algorithm VELVET to test our method and demonstrate substantial improvement in many cases. However, as we note before the algorithm running time is quadratic in the number of sequences and therefore not practical for any real world project. In this section we discuss an algorithm we develop to perform de novo clustering with short reads (Figure 1). The clusters our algorithm produces can be used as basis for any error correction method. Our method implements the MapReduce paradigm and simultaneously computes clusters of reads that have similar characteristics. We have implemented the method in Hadoop and tested it on the Amazon Elastic Cloud (EC2) infrastructure. We will show that the algorithm is suitable to process vast amount of data and that the algorithm is robust with high error rates.

MapReduce and Amazon EC2

MapReduce is a programming paradigm developed at Google Inc. The paradigm models the Lisp map/reduce functions and has two essential phases: The map phase processes the input data and outputs intermediate key/value pairs and the reduce phase takes all intermediate pairs that are associated with the same key and produces the final result. Hadoop is an open source implementation of the MapReduce model and it is becoming prominent in areas where computational tasks involve processing and generating large data sets.

Our algorithm fits the MapReduce programming model because it first divides similar reads into partitions, and each partition can be processed simultaneously for clusters with zero dependency. We selected the Amazon EC2 platform because of several advantages we can exploit. First, it is highly scalable, the Elastic Cloud environment provides a flexible way to increase or decrease the number of nodes in a cluster and the number that are dedicated to a job. Since the MapReduce framework handles data partitioning and file movement, scaling up the application is virtually effortless. Second, it is easily accessible. Because the infrastructure is set up and provided by the Amazon Elastic Cloud computing services, users are not required to have a cluster to run our application. We believe our algorithm is a contribution in the direction of using Hadoop as the platform for parallel programming in the bioinformatics space.

The Algorithm Approach

To implement the MapReduce model our method specifies a map function, which divides the input DNA sequences into groups that belong to an overlapping region (Figure 1).

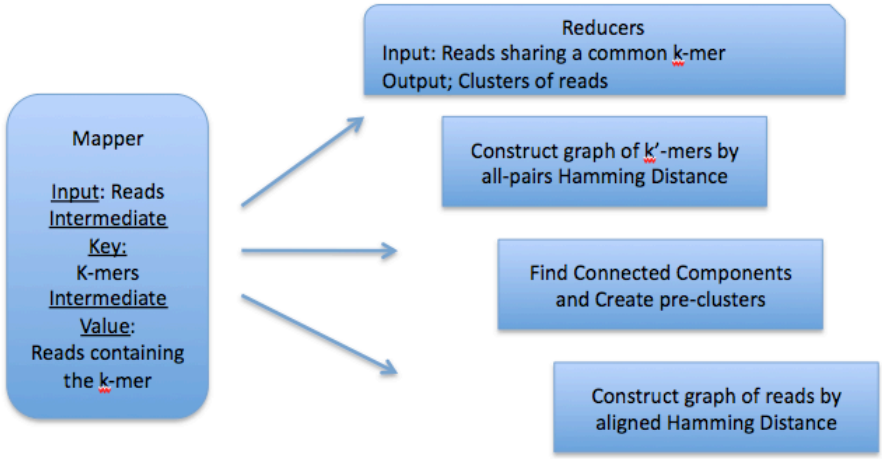


Figure 2: Algorithm overview. In our algorithm map function groups reads that overlap perfectly by a small k-mer and the reduce function refines the group of reads into clusters in parallel.

The reduce function implements a graph based algorithm and further refines the groups to produce the final read clusters (Figure 2).

Map Function

To group sequences that belong to an overlapping region, we aim to partition the data such that reads that might belong in a region together will be mapped to the same reducer. We do this by mapping the reads based on their k-mers (Figure 3). If the same k-mer is found in certain sequences, those sequences are likely to come from the same region of the genome.

For each input read, the map function extracts all shifts of a predefined length k of k-mers and for each k-mer it produces a <key, value> pair using the k-mer as key and the reads as value. Thus, sequences that share a common sub-string are grouped together and processed by the same reduce function. In the end, each read of length L appears in $L-k+1$different groups.

The correctness and efficiency of our algorithm is sensitive to k that we choose for the k-mers in this phase. As k increases, reads are mapped to more buckets, so the total input size to the reducers decreases. Also, each bucket contains fewer reads, so the computation in the reduction phase speeds up. However, when the error rate is high, a larger choice of k increases the chance that each k-mer contains errors. For a k-mer with errors the read is grouped incorrectly with others that are not from the same region of genome and not placed in the cluster it belongs to. Thus, there is a tradeoff between performance and quality, and this parameter should be carefully tuned for each data set.

Figure 3: k-mers for k=5. The figure shows how to extract all shifts of 5-mers. For the read ACCTAAGTTCATCAT, the first 3 5-mers are ACCTA, CCTAA, CTAAG.

Reduce Function

The input to each reducer is a group of reads that share a common k-mer. Since we choose k to be very small, the reads in the group may come from many different regions of the genome. We want to create clusters so that reads in the same clusters can overlap to form a single contig. To do this, we use another parameter k' where k' > k and find k'-mers that differ by less than a specified number of bases t_1, and use these k'-mers to cluster the reads together. The assumption is that reads that contain the same k'-mer, with minimal errors, will come from the same region. The first part of the reduction algorithm is to group these similar k'-mers together.

Step 1: Extract all the second-level k-mers that contain the first-level k-mer, the key for the current reduction group, from the reads. For each second-level k-mer, record all the reads it appears in as well as the index in the read. Same k-mer may appear more than once in read although very unlikely.

Step 2: Create a graph using k'-mers as nodes. Connect two k'-mers with an undirected edge if the Hamming distance between the k'-mers is less than a predefined threshold. The threshold should reflect the number of errors that are expected to appear in the k'-mer. Thus, as the maximum error rate in a data set increases, the threshold should be also be adjusted.

Step 3: Find the connected components of the k'-mer graph created in the previous step. We expect that each connected component consist of k'-mers that indeed come from the same segment in the genome, but might contain some number of errors.

At this point, we have small groups of k'-mers and each group contains k'-mers that would most likely be identical if the reads were error-free. Thus it is natural to aggregate all the reads that the k'-mers in each group appear in and output these as the read clusters. However, identical k'-mers could appear in different regions of the genome, especially when the genome is large or when repeated regions are common. Thus, we need to further refine the group of reads.

Step 1: For each group of k'-mers, aggregate all the reads that they appear in. These are called "pre-clusters". Further separate the pre-clusters in the following steps if necessary.

Step 2: Create a graph where each node represents a unique read. There is an undirected edge between two nodes if their "aligned" Hamming distance is less than a specified threshold t_2. Aligned Hamming distance is computed by aligning the two reads using the index of the k'-mers in the reads, and then computing the Hamming distance of the overlapping sections of the read. We also add another adjustable parameter that specifies the minimum amount of overlap between two reads necessary for them to be connected by an edge. Increasing this parameter reduces the cluster sizes, but ensures that the clusters have high integrity.

Step 3: Find the connected components of the graph created in step 2. Discard components smaller than a minimum threshold, since they do not give enough helpful information to correct errors in reads, and the reads contained in these components will also appear in clusters elsewhere.

The connected components found in the very last step are precisely the clusters of reads that our algorithm outputs.

Results

Clustering Accuracy

To gauge the usefulness of our cluster data for error correction applications, we mainly used two metrics to measure the quality of the clusters, cluster size and number of regions within a cluster. The cluster size is a good indicator of cluster quality. A larger cluster size provides more statistical information about the reads in it. One can estimate a theoretical limit of the cluster size to be $\frac{(L-k+1)\times c}{k}$ from our algorithm, where L is the read length and c is coverage. This limit has also proven useful in tuning parameters. A region is defined by a set of reads that can overlap to form a contig. Ideally each cluster would only contain a single region; more regions in the cluster create noise for error correction algorithms and indicate poor quality of clusters.

A useful clustering algorithm needs to be able to handle data with high error rates. Different parameters should be tuned to allow our algorithm to adapt to increasing error rates. The thresholds used in the two graph based algorithms limit the number of bases that can differ between two reads before they are placed in the same cluster, so they should be increased to allow for more errors as the rate increases.

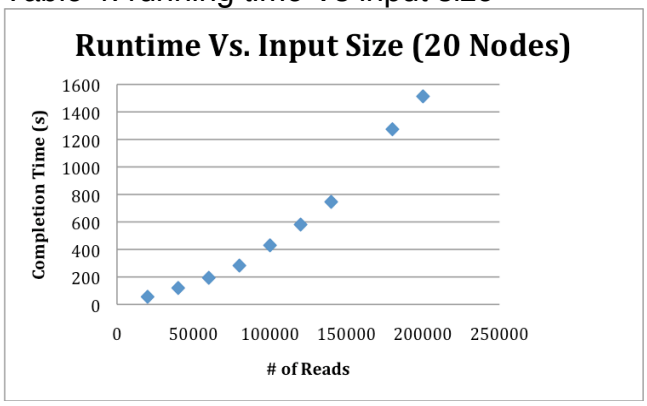
We tested our algorithm with a uniformly random genome of 50,000 bps and sampled random sequences of length 50 from it. We artificially introduced sequencing errors in the simulation at various rates. The error rate is comparable to that produced by the Illumina Genome Analyzer, which has a position-dependent error model with more errors at the end of each read that is around 6%. There are four parameters in our algorithm: k and k' are the lengths of substrings we choose to bin the reads, t_1 is the hamming distance threshold of k'-mers and t_2 is the aligned hamming distance threshold of reads. We obtained the best results with k=5, k'=15, t_1=3, t_2=6. We see that our algorithm's results are stable with increasing error rate. As we introduce more errors we only observe a small decrease in the cluster size and slight increase in number of regions per cluster (Figure 4).

Figure 4: Clustering Accuracy. Our algorithm is able to maintain the cluster size relatively well even as the error rate increases to the maximum seen in practice. Also, the mean number of regions per cluster increases by a negligible amount, which means that cluster integrity is not compromised.

Performance Analysis

We conducted these experiments using Amazon's Elastic MapReduce service. We performed the test using 20 instances with 15GB of RAM and 4 virtual cores per instance. The number of reduce tasks is set to 76 following the recommended setting of 0.95* (number of nodes) * (number of cores per node). It is important to note that each reducer runs an algorithm with $O(n^2)$ in of space and running time, n is the number of sequences to be clustered. Our data reflects this fact. For instance, as the input size doubles from 100,000 to 200,000 reads, the completion time is roughly 4 times longer (Table 4). In these experiments, we were limited to use only 20 nodes, which is the maximum allowed by our Amazon account. However the results strongly indicate that as soon as more nodes become available we can process much larger datasets efficiently. It is also important to note that our sequential implementation of the algorithm failed to complete within a reasonable amount of time on an input of size 20,000.

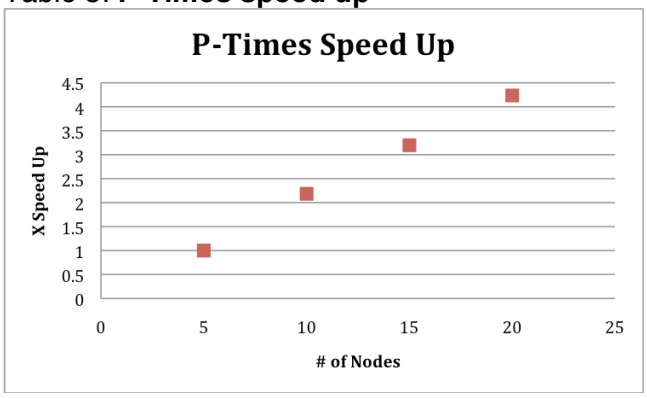
Table 4: running time Vs input size



# of Reads	Completion Time (sec)
20000	56
40000	120
60000	194
80000	283
100000	430
120000	581
140000	746
180000	1274
200000	1513

Table 4 summarizes how the algorithm scales with input size.

Table 5: **P-Times speed up**



Our method has ideal P-Times speed up characteristics

# of Reads	Completion Time (sec)
20000	56
40000	120
60000	194
80000	283
100000	430
120000	581
140000	746
180000	1274
200000	1513

P-Times Speed Up and Load Balance

An important metric in measuring parallel programs is P-Times speed up which examines the runtime as function of number of CPUs. In this experiment we ran the application on an input dataset of 80,000 reads of length 50 base pairs sampled from a random genome. We recorded the completion time using different numbers of nodes. Our experiments show that it very closely approximates the idealized p-times speedup as we allocate more nodes for the job (Table 5). This is because we have a very large number of reduce jobs, and as we increase the number of nodes the reduce jobs are simply split among the increased resources since there is no communication overhead. We used 5 nodes as a base for the speedup because even for a moderate reads input size of 80,000, the sequential version of our algorithm takes far too long to complete. Figure 5 and Table 2 show that as we increase the number of nodes from 5 to 20, we have successfully reached a 4X speed up. This illustrates that as we expected, using the Hadoop platform in conjunction with cloud computing allows our code to be highly scalable. In practice, our data lead us to believe that large input sets can be handled by adding more nodes, which is a major advantage of cloud computing. To measure load balancing we investigated the aspect of load balancing that is inherently part of the data: the number of unique sequences that contain the same substring. We simulated the map function and calculated the size and standard deviation of each reduce task input (Table 6).

The input file is a simulated random reads file from a uniformly random genome. The results suggest that the choice of k size places a significant role in the amount of computational work to be done by each reduce task and should be studied with care.

Discussion

In this section we present a highly scalable distributed algorithm for solving the short-reads clustering problem. Our method uses graph-based techniques to cluster DNA sequences that overlap. The algorithm is robust in dealing with data with varied error rates; our results show that increased error rate in data has little effect on the quality of the clusters the algorithm produces.

Table 6: **Load Balance**

k	Avg Reads / Key	Standard Dev (s)
5	3593	280.6
7	214	39.93
9	12	8.3
11	2	2.71

The column labeled “Avg Reads/Key” corresponds to the size of workload in each node in reduce phase. The table suggests that the work load varies greatly with the choice of k and this will be a focal point in future studies.

We implement the algorithm in Hadoop (http://www.hadoop.org), which is an open-source Java implementation of the MapReduce model. We test our implementation using Amazon's Elastic MapReduce on the Amazon Elastic Cloud Compute infrastructure and obtain encouraging results. The method we develop is an early stage attempt at the solution, although we have demonstrated the potential of running graph based clustering in parallel on Hadoop there still remains many opportunities to improve and explore. One of the interesting areas is to parallelize the reduce function even further by using a consequent MapReduce job. Our current implementation must refine many pre-clusters of reads in sequence for each reduce task, if we parallelize this process and refine each pre-cluster in a separate machine we can potentially achieve better performance. Another area worth exploring is to use the MapReduce method do consensus voting error correction.

4. Conclusion

In this report we present our preliminary work in the study of de novo assembly with short-reads. We study the current assembly algorithms and identify two major areas where they can be improved and conducted experiments on our propose solutions. The first area is to harvest quality information in the reads. In order to show that quality scores contain rich information we modify popular software VELVET to incorporate reads quality scores in its Tourbus algorithm. We experiment two versions of the VELVET program on simulated sequence data with various coverage depths and under three different error profiles. The results show that VELVET with quality scores can achieve substantially higher accuracy than using the reads alone in all cases (as much as 75% improvement).

The other area of focus for us is error correction preprocessing. Many de novo assembly algorithms today are very sensitive to sequencing errors and although most of them make use of error correction techniques but they are mostly ad hoc and are not well understood. We propos an error correction method based on k-mer graphs and it can be used in a preprocessing phase to clean up data for any assembly algorithm. We test our method on VELVET and observe a significant boost in accuracy and percentage of bases assembled. It is worth noting that our error correction method is particularly effective in regions of low coverage. We think that the error correction

algorithm can help today's assemblers to achieve better performance with lower coverage requirement and thus reduce the cost of large-scale sequence projects. Our last aim in this study is then to develop a scalable and robust de novo clustering method with short reads to help massive scale sequencing projects. We develop a parallel algorithm that uses graph theory based techniques to cluster reads and demonstrate that the algorithm has excellent robustness and scalability. A direct application of our clustering method is to apply statistical error correction methods on the clusters produced by our method. The scalability of our clustering method can enable error correction on very large data sets.

Acknowledgements

I want to dedicate my thanks to Professor Satish Rao and Professor Yun Song their guidance and mentorship throughout this study. I thank Wei-Chun Kao for valuable input on the clustering algorithm. I thank Matei Zaharia for helping get started on Amazon MapReduce and providing answers to numerous related questions. I also thank Aileen Chen and Jerry Hong for their dedication on implementing solutions and conducting experiments.

Reference

[1] Butler J, Maccallum I, Kleber M, Shlyakhter IA, Belmonte MK, Lander ES, Nusbaum C, Jaffe DB. 2008. ALLPATHS: de novo assembly of whole-genome shotgun microreads. Genome Res. 18:810–820.

[2] Chaisson,M.J. Dumitru Brinza, and Pavel A. Pevzner (2009) *De novo* fragment assembly with short mate-paired reads: Does the read length matter? *Genome Res.*, 19, 336–346.

[3] Hernandez, D., Francois, P., Farinelli, L., Osteras, M. & Schrenzel, J. De novo bacterial genome sequencing: Millions of very short reads assembled on a desktop computer. Genome Res. 18, 802–809 (2008).

[4] Mihai Pop. Genome assembly reborn: recent computational challenges. Brief Bioinform, 10(4):354:366, July 2009.

[5] Warren, R.L., Sutton, G.G., Jones, S.J.M., and Holt, R.A. 2007. Assembling millions of short DNA sequences using SSAKE. Bioinformatics 4: 500–501.

[6] Zerbino DR, Birney E. 2008. Velvet: algorithms for de novo short read assembly using de Bruijn graphs. Genome Res. 18:821–829.