White Paper

White paper on de novo assembly in CLC Assembly Cell *3.0*

March 16, 2010

CLC bio
Finlandsgade 10-12 · 8200 Aarhus N · Denmark
Telephone: +45 70 22 55 09 · Fax: +45 70 22 55 19
www.clcbio.com · info@clcbio.com

White Paper

Contents

White Paper

1 Introduction

This is a white paper on the *de novo* assembler in CLC Assembly Cell *3.0*. Note that the same algorithm is used by *CLC Genomics Workbench* and *CLC Genomics Server*, so except for the performance benchmarks (speed and memory), this white paper applies to these products as well after next release.

The assembler is designed to combine a mix of data from Illumina, 454, SOLiD and Sanger sequencing, both as single and as paired-end reads. For paired-end data, different insert sizes can be combined in the same assembly. Note that in the current version, paired-end SOLiD data can be used in a post-processing step to link contigs together.

This white paper consists of three parts: The first part explains how the assembler works, and the next focuses on large genome assembly of a human data set, where we have compared our assembler with the ABySS assembler which is also capable of assembling human genome-size data sets [Simpson et al., 2009]. The third part reports the results of a smaller bacterial data set where focus is on quality and where we have compared the quality and performance of our own assembly with one of the popular open source assembly algorithms, *Velvet* [Zerbino and Birney, 2008].

2 How it works

CLC bio's de novo assembly algorithm works by using de Bruijn graphs. This is similar to how most new de novo assembly algorithms work. The basic idea is to make a table of all sub-sequences of a certain length (called words) found in the reads. The words are relatively short, e.g. about 20 for a bacterial genome and 27 for a human genome.

Given a word in the table, we can look up all the potential neighboring words (in all the examples here, word of length 16 are used) as shown in figure 1.

Backward neighbors	Starting word	Forward neighbors
AACGTAGCTAGCGCAT	ACGTAGCTAGCGCATG	CGTAGCTAGCGCATGA
CACGTAGCTAGCGCAT		CGTAGCTAGCGCATGC
GACGTAGCTAGCGCAT		CGTAGCTAGCGCATGG
TACGTAGCTAGCGCAT		CGTAGCTAGCGCATGT

Figure 1: *The word in the middle is 16 bases long, and it shares the 15 first bases with the backward neighboring word and the last 15 bases with the forward neighboring word.*

Typically, only one of the backward neighbors and one of the forward neighbors will be present in the table. A graph can then be made where each node is a word that is present in the table and edges connect nodes that are neighbors. This is called a de Bruijn graph.

For genomic regions without repeats or sequencing errors, we get long linear stretches of connected nodes. We may choose to reduce such stretches of nodes with only one backward and one forward neighbor into nodes representing sub-sequences longer than the initial words.

Figure 2 shows an example where one node has two forward neighbors:

After reduction, the three first nodes are merged, and the two sets of forward neighboring nodes are also merged as shown in figure 3.

Figure 2: *Three nodes connected, each sharing 15 bases with its neighboring node and ending with two forward neighbors.*

Figure 3: *The five nodes are compacted into three. Note that the first node is now 18 bases and the second nodes are each 17 bases.*

So bifurcations in the graph leads to separate nodes. In this case we get a total of three nodes after the reduction. Note that neighboring nodes still have an overlap (in this case 15 nucleotides since the word length is 16).

Given this way of representing the de Bruijn graph for the reads, we can consider some different situations:

When we have a *SNP or a sequencing error*, we get a so-called bubble as shown in figure 4.

Figure 4: *A bubble caused by a SNP or a sequencing error.*

Here, the central position may be either a C or a G. If this was a sequencing error occurring only once, we would see that one path through the bubble will only be words seen a single time. On the other hand if this was a SNP we would see both paths represented more or less equally. Thus, having information about how many times this particular word is seen in all the reads is very useful and this information is stored in the initial word table together with the words.

If we have a *repeat sequence* that is present twice in the genome, we would get a graph as shown in figure 5.

Figure 5: *The central node represents the repeat region that is represented twice in the genome. The neighboring nodes represent the flanking regions of this repeat in the genome.*

Note that this repeat is 57 nucleotides long (the length of the sub-sequence in the central node above plus regions into the neighboring nodes where the sequences are identical). If the repeat had been shorter than 15 nucleotides, it would not have shown up as a repeat at all since the word length is 16. This is an argument for using long words in the word table. On the other hand, the longer the word, the more words from a read are affected by a sequencing error. Also, for each extra nucleotide in the words, we get one less word from each read. This is in particular an issue for very short reads. For example, if the read length is 35, we get 16 words out of each read of the word length is 20. If the word length is 25, we get only 11 words from each read.

To strike a balance, CLC bio's de novo assembler chooses a word length based on the amount of input data: the more data, the longer the word length.

A simple de novo assembly result would be to output the sequence of each reduced node. The bubbles described above from SNPs and sequencing errors as well as the repeats will make this quite a bad result with many short contigs. Instead, we can try to resolve the repeats with

reads that span from a node before the repeat to a node after the repeat. Small bubbles can be resolved by choosing the path with the most coverage. Thus, by using the information from the full length reads, we are able to produce much longer contigs.

Furthermore, when paired reads are available, we can use this information to resolve even larger repeat regions that may not be spanned by individual reads, but are spanned by read pairs. This results in even longer contigs.

So in summary, the de novo assembly algorithm goes through these stages:

- Make a table of the words seen in the reads.
- Build de Bruijn graph from the word table.
- Use the reads to resolve the repeats.
- Use the information from paired reads to resolve larger repeats.
- Output resulting contigs based on the paths.

These stages are all performed by the assembler program.

Repeat regions in large genomes often get very complex: a repeat may be found thousands of times and part of one repeat may also be part of another repeat, further complicating the graph. Sometimes a repeat is longer than the read length (or the paired end distance when pairs are available) and then it becomes impossible to resolve the repeat. This is simply because there is no information available about how to connect the nodes before the repeat to the nodes after the repeat. This means that no matter how much coverage we have, we will still get a number of separate contigs as a result.

3 SOLiD data support in de novo assembly

SOLiD sequencing is done in color space. When viewed in nucleotide space this means that a single sequencing error changes the remainder of the read. An example read is shown in figure 6.

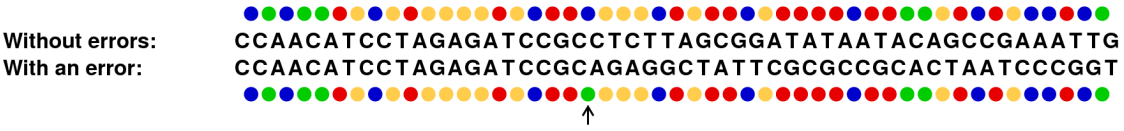

Figure 6: *How an error in color space leads to a phase shift and subsequent problems for the rest of the read sequence*

Basically, this color error means that C's become A's and A's become C's. Likewise for G's and T's. For the three different types of errors, we get three different ends of the read. Along with the correct reads, we may get four different versions of the original genome due to errors. So if SOLiD reads are just regarded in nucleotide space, we get four different contig sequences with jumps from one to another every time there is a sequencing error.

Thus, to fully accommodate SOLiD sequencing data, the special nature of the technology has to be considered in every step of the assembly algorithm. Furthermore, SOLiD reads are fairly short and often quite error prone. Due to these issues, we have chosen not to include SOLiD support

in the first algorithm steps, but only use the SOLiD data where they have a large positive effect on the assembly process: when applying paired information. Thus, the `clc_novo_assemble` program of the CLC Assembly Cell has a special option ("-p d") to indicate that a certain data set should be used only for their paired information[1]. This option should always be applied to SOLiD data. It is also useful for data sets of other types with many errors. The errors might have the effect of confusing the initial graph building more than improving it. But the paired information is still valuable and can be used with this option.

4 Measuring quality

Before presenting the benchmarks, we now introduce the measures used. The benchmarks primarily focus on measuring the quality of the assembly. It is not easy to construct one single accurate measure of quality, so we use a combination of measures that show different aspects of quality:

Number of contigs The number of contigs produced. This is usually not a number you can use to assess general quality.

Incorrect contigs The number of contigs that are incorrect. The data sets that we have tested have a known reference sequence that we can compare the contigs with. A contig is counted as "correct" when at least 95 % of the contig match the reference sequence (we use BLAST to test this). An **example**: if you have a contig of 5000 bases where 2500 bases match perfectly one place on the genome and the other 2500 match perfectly another place, the whole contig will be counted as incorrect. If the number of incorrect contigs is small, this number cannot really be used since there is, even for this data set, a few variations when comparing with the reference sequence. Note that this is only done for the bacterial data set.

Sum Counts the number of bases when all contigs are put together.

Contig sizes Counts the maximum and average contig lengths.

N50 This is a standard measure for *de novo* assembly. It is a better way of measuring the lengths of contigs. It is calculated by summarizing the lengths of the biggest contigs until you reach half the size of the sum of all contig lengths. The N50 is then the size of the contig that was added last (i.e. the smallest of the big contigs covering 50 % of the summed contig lengths).

Of these measures, the most important ones are the correct/incorrect measure and the N50. Coverage is not so important, since there is not much difference between the programs here. Note that these numbers are also interrelated: if you have large contigs, you will have a large N50 value, but there is also a greater risk that some of them contain errors.

5 Testing performance – assembling a human genome

The data set used for the human genome assembly is produced with Illumina's Genome Analyzer and consists of approximately 3.6 billion reads resulting in 38 fold coverage of the human genome. Most reads have a length of 36 bp, summing up to 130 Gbp in total.

[1] There is a similar option in the *CLC Genomics Workbench* and *CLC Genomics Server*

The data consists of 606 files in fastq format. The file size is 138 GB in gzip format. The data is read directly from the zipped files.

We have not performed the benchmarks for the ABySS assembler ourselves - they have been reported in Simpson et al., 2009.

5.1 Without paired-end information

The results of assembling the data as single reads is shown in table 1.

	CLC	ABySS
Number of computers	**1**	**21**
CPUs		
CPU cores per computer	8	8
Total CPU cores	8	168
Memory		
RAM per computer (GB)	72	16
Total RAM usage (GB)	21	336
Assembly time		
Time spent (hours)	6	15
Contigs >= 100 bp		
Number of contigs	3,781,869	4,348,132
Mean size (bp)	584	484
Max size (bp)	18,222	15,911
N50 size (bp)	1,156	871
Sum (Gbp)	2.2	2.1
Contigs >= 1000 bp		
Number of contigs	638,661	549,522
Mean size (bp)	1,918	1,703
Max size (bp)	18,222	15,911
N50 size (bp)	2,021	1,731
Sum (Gbp)	1.2	0.9

Table 1: Benchmark of the Illumina data set as un-paired (both ABySS and CLC).

The most significant difference is in speed and hardware requirements. The CLC assembler is running on one powerful workstation computer with 72GB RAM, the ABySS has been run on a cluster of 21 computers with 16 GB RAM each. Even though the hardware set-up is much smaller, the single machine running the CLC assembler only uses 7 hours versus the 15 hours of the cluster running ABySS.

Looking at the results, the N50 is higher for the CLC assembler (1,156 vs 871). This means that the CLC assembler produces more long contigs than ABySS. This is also visible when looking at the fraction of the genome covered by contigs longer than 1,000 bp. The CLC assembler covers 1.2 Gbp whereas ABySS covers 0.9 GBP.

5.2 Making use of paired-end information

When the paired-end information is added, you will expect the result to be better, i.e. to get longer contigs. The results of assembling the paired-end data set is shown in table 2. Note that it is the same data set as above, but this time the assembler knows how the reads are paired.

	CLC	**ABySS**
Number of computers	**1**	**21**
CPUs		
CPU cores per computer	8	8
Total CPU cores	8	168
Memory		
RAM per computer (GB)	72	16
Total RAM usage (GB)	42	336
Assembly time		
Time spent (hours)	7	80
Contigs >= 100 bp		
Number of contigs	3,313,745	2,762,173
Mean size (bp)	665	791
Max size (bp)	32,285	18,800
N50 size (bp)	1,410	1,499
Sum (Gbp)	2.2	2.2
Contigs >= 1000 bp		
Number of contigs	655,413	680,230
Mean size (bp)	2,090	2,093
Max size (bp)	32,285	18,800
N50 size (bp)	2,273	2,282
Sum (Gbp)	1.4	1.4

Table 2: Benchmark of the Illumina data set as paired (both ABySS and CLC).

In terms of speed, you can see that the CLC assembler is completing the analysis approximately using the same time as with the single reads. The processing time for the ABySS assembler increases from 15 to 80 hours.

In terms of the results, then the N50 is going up, especially for ABySS which now has an N50 comparable with the one of the CLC assembler. You can see that the effect of adding the paired-end information for the CLC assembler is that N50 goes from 1,156 to 1,410, and you now get 1.4 Gbp of the genome covered with contigs longer than 1,000 bp (versus 1.2 Gbp without paired-end information).

5.3 Assembling long reads

In addition to the benchmarks above, we have tested another human data set produced on the 454 platform. This is the Watson genome [Wheeler et al., 2008].

The sequencing data consists of 76,526,397 reads which gives 19,243,100,298 bp all in all. This means that there is a coverage level around 6 times.

	CLC
Number of computers	**1**
Total CPU cores	8
Memory	
Peak RAM usage (GB)	19.1
Assembly time	
Time spent (hours:minutes)	3:22
Contigs >= 200 bp	
Number of contigs	1,874,102
Mean size (bp)	1,368
Max size (bp)	25,831
N50 size (bp)	2,267
Sum (Gbp)	2.6

Table 3: Benchmark of the 454 Watson genome.

The assembler used 3 hours and 22 minutes assembling this data set. The peak of memory usage was at 19.1 GB. We have not compare the results with other assemblers.

Note that increasing coverage will most likely improve results considerably.

6 Testing quality on a smaller data set

The data sets used for the benchmarks are shown in table 4.

The first *E. Coli* data set is sequenced on the 454 platform ($\tilde{2}$35 bp reads) and has about 19 times coverage of the genome. The second data set comes from an Illumina Genome Analyzer

Name	Reference (bp)	Reads	Reads (bp)	Length
454 *E. Coli*	4,639,675	436,142	102,475,824	Long (~235 bp)
Illumina *E. Coli*	4,639,675	5,244,764	183,566,740	Short (35 bp)

Table 4: The data sets used for benchmarks.

(35 bp reads) and has about 16 times coverage of the genome. They are both commensal strain K-12 of *E. Coli*.

We compare the the latest version of *Velvet* (0.7.55) with the *de novo* assembly in the CLC Assembly Cell *3.0*. We have not measured the speed and memory consumption of the two algorithms because the data sets are so small. Both sets are assembled in a couple of minutes on a fast computer.

6.1 454 data set (long reads)

The results for the 454 data set are shown in table 5. Both the results of CLC bio's old de novo assembler (available in CLC Assembly Cell 2 and *CLC Genomics Workbench* 3), the new beta version and Velvet are reported. Note that only contigs longer than 500 bp are included.

	CLC, old	CLC, new	Velvet
Contigs	95	130	264
Incorrect	14	4	1
Sum (bp)	4,702,324	4,552,181	4,496,998
Contig size			
Max	267,932	213,175	119,550
Average	49,498	35,017	17,034
N50	100,145	58,886	29,435

Table 5: Benchmark of the 454 data set, reported for contigs longer than 500 bp.

There is a significant difference in the lengths of the contigs created by the two algorithms. *Velvet* produces contigs with a N50 of 29,435, whereas CLC produces contigs with a N50 of 58,886.

6.2 Illumina data set unpaired (35 bp)

The Illumina data set has much shorter reads (35 bp) and shows a different pattern. We have run the same data set twice: one using paired-ends information and one where the reads are treated as single reads. Results from the run without paired-ends are shown in table 6.

The conclusion of this run is that *Velvet* this time produces a better result than the new CLC assembler. The N50 is higher and the average contig size is bigger. The old CLC assembler is better when it comes to contig length, but it also creates a high number of incorrect contigs.

	CLC, old	CLC, new	Velvet
Contigs	374	586	474
Incorrect	36	0	1
Sum (bp)	4,611,538	4,516,884	4,502,310
Contig size			
Max	69,266	55,700	60,572
Average	12,330	7,708	9,499
N50	23,335	12,655	16,408

Table 6: Benchmark of the Illumina data set as un-paired.

6.3 Illumina data set paired

Running the same data set, this time using paired-ends information, gives the results shown in table 7.

	CLC, old	CLC, new	Velvet
Contigs	161	350	117
Incorrect	13	0	43
Sum (bp)	4,638,306	4,528,612	4,529,359
Contig size			
Max	155,212	69,484	174,035
Average	28,809	12,939	38,712
N50	51,315	22,910	95,356

Table 7: Benchmark of the Illumina data set as paired.

With the paired-end data set, the new CLC assembler produces significantly smaller contigs than both the old CLC assembler and Velvet. But the long contigs are also accompanied by a lot of incorrect contigs, both by Velvet and the old CLC assembler.

We should note that you can probably achieve better performance for Velvet by spending more time adjusting the numerous parameters to fit this particular data set. With the CLC assembler, there are no parameters to set to affect the way the assembly works (except defining the paired-end read distances).

7 Conclusion

Based on the benchmark results above, it is not possible to conclude which algorithm is the best. They each have their strengths and weaknesses.

For the big data set, the difference in results between ABySS and the CLC assembler is minimal, but when it comes to hardware requirements, speed and memory consumption, the CLC assembler performs better than ABySS. The ABySS benchmark is run on a 21 node cluster and

the CLC assembler is tested on one computer. For the paired-end data set, ABySS spends 80 hours whereas the CLC assembler spends 7 hours.

We used the small data set to get an impression of the quality of the results. In general, there is a trade-off between having long contigs and few errors. This is the case for both the CLC assembler and *Velvet*. The use context plus the nature of the data will determine what it is the best solution. With this white paper we have shown what it looks like in a direct comparison. What we have *not* compared is the user interface and output of the two solutions. To use *Velvet*, you have to download and compile the code on your computer before running it (from the command line). The CLC Workbench comes as a desktop application with a graphical user interface or in a server set-up. Alternatively, you can use the CLC assembler from the command line as part of the *CLC Assembly Cell*.

References

[Simpson et al., 2009] Simpson, J. T., Wong, K., Jackman, S. D., Schein, J. E., Jones, S. J. M., and Birol, I. (2009). Abyss: A parallel assembler for short read sequence data. *Genome Res*, 19(6):1117–1123.

[Wheeler et al., 2008] Wheeler, D. A., Srinivasan, M., Egholm, M., Shen, Y., Chen, L., McGuire, A., He, W., Chen, Y.-J., Makhijani, V., Roth, G. T., Gomes, X., Tartaro, K., Niazi, F., Turcotte, C. L., Irzyk, G. P., Lupski, J. R., Chinault, C., zhi Song, X., Liu, Y., Yuan, Y., Nazareth, L., Qin, X., Muzny, D. M., Margulies, M., Weinstock, G. M., Gibbs, R. A., and Rothberg, J. M. (2008). The complete genome of an individual by massively parallel dna sequencing. *Nature*, 452(7189):872–876.

[Zerbino and Birney, 2008] Zerbino, D. R. and Birney, E. (2008). Velvet: algorithms for de novo short read assembly using de Bruijn graphs. *Genome Res*, 18(5):821–829.