Parallel and memory-efficient reads indexing for genome assembly

Guillaume Chapuis, Rayan Chikhi, Dominique Lavenier

Computer Science department, ENS Cachan/IRISA, 35042 Rennes, France

Abstract. As genomes, transcriptomes and meta-genomes are being sequenced at a faster pace than ever, there is a pressing need for efficient genome assembly methods. Two practical issues in assembly are heavy memory usage and long execution time during the read indexing phase. In this article, a parallel and memory-efficient method is proposed for reads indexing prior to assembly. Specifically, a hash-based structure that stores a reduced amount of read information is designed. Erroneous entries are filtered on the fly during index construction. A prototype implementation has been designed and applied to actual Illumina short reads. Benchmark evaluation shows that this indexing method requires significantly less memory than those from popular assemblers.

Introduction

Until the emergence of next-generation sequencing (NGS) technologies, software for assembling genomes could process up to millions of long ($\sim 10^4$ bp) reads. Now, a typical genome assembly instance for a vertebrate genome consists of billions of short (100 bp) reads. Despite this technological shift, computational models for assembly are essentially based on constructing and simplifying a genome graph. However, graph-based models have inherent limitations that make them unpractical for assembly of NGS data. They require the construction of either a large string graph containing all the reads, or a de Bruijn graph containing all the k-length substrings (k-mers) of the reads. For human-sized genomes, the de Bruijn graph typically requires hundreds of gigabases of memory [9]. Nevertheless, NGS assembly tools rely on optimized implementations of these graph models. For instance, leading assembly programs have implemented efficient heuristics using a de Bruijn graph [18,9]. For more details concerning these implementations, refer to a recent survey [10]. In a near future, larger eukaryotic genomes and meta-genomes will be sequenced at a faster pace than computational resources growth. Hence, new assembly models need to be developed to sustain the increasing rate of NGS technologies.

Several theoretical advances have been recently proposed to reduce the memory usage of graph-based assemblers. Simpson et al. implemented compression techniques (FM-index [6]) during construction of the string graph [15] at the expense of running time. Conway et al. used succinct bitmap structures [11,13] to construct an immutable de Bruijn graph [5]. Distributed de Bruijn graph construction using a message passing interface have been implemented in several assemblers [7,16,8].

Greedy assemblers use a different assembly strategy. Instead of constructing a genome graph, they repeatedly perform an extension procedure until branching is detected. Previous implementations of greedy assemblers used a prefix tree to store reads [17], which consumes significantly more memory than a de Bruijn graph. Recent optimized implementations use custom k-mer indexing structures for memory efficiency [1,2,3]. In particular, these implementation have been applied to complex mammalian genomes, demonstrating that greedy assemblers are not limited to genomes with low repeat content. Unlike de Bruijn graph assemblers, data structures used in greedy assemblers typically contain references to read sequences. Hence, efficient read indexing is necessary to keep memory usage low.

In the next section, we propose a parallel reads indexing procedure designed specifically for assembly. Two novel filtering methods are introduced to reduce memory usage: a procedure to remove erroneous k-mers on the fly, and a procedure to avoid referencing redundant reads. Finally, a prototype implementation is applied to real Illumina data to validate the method.

Methods

Distributed and multi-threaded indexing

A multi-threaded, multi-node procedure for reads indexing is proposed. A hash table is constructed, where the entries are k-mers, and the values are references to reads. Taking advantage of shared memory between threads, reads sequences are stored separately in memory, without redundancy within a node. Index construction is distributed among N nodes, and each node performs independent computations in parallel. Specifically, each node n is running T_n threads, each thread t_n constructs a separate sub-index $I(n, t_n)$. A binning method adapted from [16] assigns each k-mer to a unique sub-index. Let h be a k-mer hash value with perfect hashing [16]. The corresponding k-mer belongs to the sub-index $I(n, t_n)$ if:

$$\begin{cases} h \ mod \ N = n \\ h \ mod \ T_n = t_n \end{cases}$$

which ensures that each sub-index contains distinct k-mers. Each thread reads the entirety of the input data to construct its sub-index. When all the sub-indexes are constructed, an inexpensive merging phase yields the complete index. Hence, the indexing procedure always constructs the same complete index on different architectures. In the following, two algorithmic ingredients are described for parallel sub-index construction: k-mers filtering and reads indexing.

On-line parallel k-mers filtering

Memory efficiency is crucial when assembling NGS data. In many approaches, including the one proposed here, memory consumption is proportional to the number of indexed k-mers. It is therefore important to filter out erroneous k-mers as early in the indexing process as possible. Erroneous k-mers are produced whenever the sequencing process makes a mistake during base calling. The abundance distribution $K_t^n(m)$ is defined as the number of k-mers seen exactly m times at indexing time t by node n. A key fact is that the hash function used above evenly distributes k-mers among sub-indexes. Hence, each $K_t^n(m)$ is identically distributed as the entire distribution $\sum_n K_t^n(m)$. This observation enables independent, parallel filtering for each sub-index. The superscript n is then omitted in the following.

A typical distribution of $K_t(m)$ at final time t is multimodal. A large number of k-mers occur only a few times: these are mostly sequencing errors. Assuming uniform sequencing coverage, the distribution of correct k-mers is a Gaussian mixture. The most abundant component is centered at the expected coverage of the target genome. Less abundant components are centered at multiples of the coverage, due to repeats in the genome. The proposed method consists in (i) detecting components corresponding to erroneous and correct k-mers as soon as they separate sufficiently from each other and (ii) finding an appropriate erroneous threshold (cut-off value). Every k-mer that has appeared fewer times than the erroneous threshold so far is then considered as an error and removed. This procedure could be extended to correct errors in reads, but it is outside the scope of the current indexing scheme.

Error detection The following two inequalities must be satisfied to trigger the filtering procedure. First, erroneous k-mers are identified by their abundance. Theorem 3 from [12] establishes that, under reasonable sequencing assumptions, an error is significantly less likely to appear $m + 1$ times than m times. Thus, the abundance of erroneous k-mers peaks at $m = 1$ and has a strictly decreasing slope. The low end $m_{low}(t)$ is computed as the largest m that satisfies $K_t(m - 1) > K_t(m)$ for $m \geq 1$. Then, the peak abundance $m_{high}(t)$ of correct k-mers is computed as the parameter at which the maximum value of $K_t(m)$ is attained for $m > m_{low}(t)$. Erroneous and correct k-mers are considered to be separated when:

$$m_{high}(t) - m_{low}(t) > r$$

where r is a user-defined resolution parameter. Second, to avoid the computational cost of filtering too soon or too often, a constraint is imposed on the amount of erroneous k-mers. Let S_{min} be a minimum amount (user-defined) of erroneous k-mers before the filtering process can be performed:

$$\sum_{m=1}^{m_{low}} K_t(m) > S_{min}$$

Calculating the cut-off value During early filtering passes, a small fraction of correct k-mers still contributes to the erroneous component. Hence, removing the entire component at each filtering pass is not a sensible choice. An incrementing value, defined as the *cut-off value*, is introduced to overcome this problem. All k-mers of abundance lower than the cut-off value are removed by the filtering procedure, others are kept. Formally, a threshold m_{solid} is defined as the number of occurrences below which a k-mer is considered a potential error. All k-mers over this threshold at the end of the indexing phase are *solid* k-mers. Let $tReads$ and $nReads(t)$ be the total number of reads in the input file and the number of reads processed at time t respectively. The cut-off value $F(t)$ is calculated according to the following formula:

$$F(t) = \lfloor \frac{m_{solid} \cdot nReads(t)}{tReads} \rfloor$$

Reads indexing structure

Each sub-index is populated independently with a filtered set of references to reads, given a filtering function designed for *de novo* assembly. The *extension* of a k-mer in a read is defined as the suffix immediately following the k-mer (e.g. for a read $r = uwv$ where w is a k-mer and u, v are arbitrary strings, v is the extension of w in r). We introduce a notion of redundancy between extensions. Let (v_1, v_2) be two extensions of the same k-mer, without loss of generality assume that the length $|v_1|$ is shorter than $|v_2|$. Two extensions $v_{1,}v_2$ are said to be t-redundant if the Hamming distance between their prefixes of length $|v_1|$ is lower than t. The *representative read spectrum* with similarity threshold t, noted $RRS(k, t)$, is defined for a set of input reads as follows:

(i) associate a set S_w to each solid k-mer w occurring in the reads
(ii) S_w discards all but one of the reads associated to t-redundant extensions. A read with the longest extension is kept, ties are broken arbitrarily.

Figure 1 shows an example of a representative reads spectrum. The reads sequences referenced by the RRS are stored separately. Practically, both a read and its reverse-complement are indexed. References to paired-end reads are explicitly made to the left or the right mate of the read.

In essence, this structure records a representative set of reads for each solid k-mer. Note that this indexing does not correct errors in read, but merely ignores errors in reads suffixes. Erroneous prefixes yield un-solid k-mers, hence these reads are not indexed in the structure. This property is well suited with Illumina reads as sequencing errors are known to mostly occur at read suffixes. Provided the sequencing coverage is high, errors in suffixes can be corrected at a later stage during a consensus phase. This justifies the arbitrary removal of other reads having equally long t-redundant extensions. To maximize the effectiveness of the structure for assembly, sequencing reads should contain solid k-mers corresponding to every position in the genome. Hence, either a high sequencing

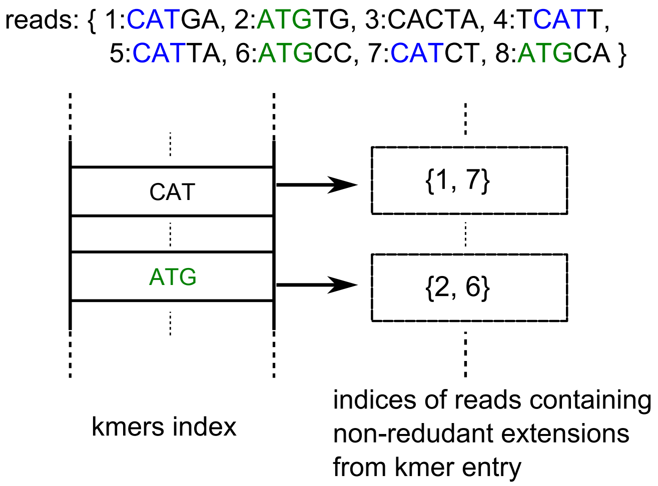
reads: { 1:CATGA, 2:ATGTG, 3:CACTA, 4:TCATT,
5:CATTA, 6:ATGCC, 7:CATCT, 8:ATGCA }
CAT
ATG
{1, 7}
{2, 6}
kmers index
indices of reads containing non-redudant extensions from kmer entry

Fig. 1. The representative reads spectrum for a set of 8 reads with parameters $k = 3, t = 1$. Entries are solid k-mers from reads. Each k-mer is associated with a list of reads which extend the k-mer to the right. The extensions are filtered for t-redundancy. For instance, reads 4 and 5 are not indexed to the CAT entry because extensions T and TA are 1-redundant with respect to extension GA from read 1. Reverse complements of reads are also indexed, but are omitted in this figure.

coverage or a low error-rate is required. Both criteria are typically met with recent Illumina sequencers.

For assembly, it can be verified that basic traversal of a string graph can be performed with this structure. The RRS acts as an incomplete inverted index for the reads. Specifically, in the string graph, out-neighbors of a read (i.e., other reads that overlap that read to the right) are retrieved from the RRS by querying each of the read k-mers. In-neighbors (left overlap) are equivalent to out-neighbors of the read reverse complement.

Results

We developed an implementation of the on-line k-mers filtering and the reads indexing algorithms, as part of the Monument assembler [4]. The implementation is tested on two actual sequence datasets from *R. sphaeroides* (SRA reference SRR034530) and *N.crassa* (all libraries from [14]) sequenced using the Illumina technology. The *R. sphaeroides* dataset (dataset 1) contains 46 million reads of length 36 bp. The *N.crassa* dataset (dataset 2) contains 320 million reads of average length 32 bp. Benchmarks were run on a 64-bit 8-cores machine with 66 GB of memory. In this implementation, read sequences are stored in memory on each node as an array of 2-bit encoded sequences. In the case of multi-nodes computation, $\frac{n}{4}$ bytes are redundantly stored per node, where n is the number

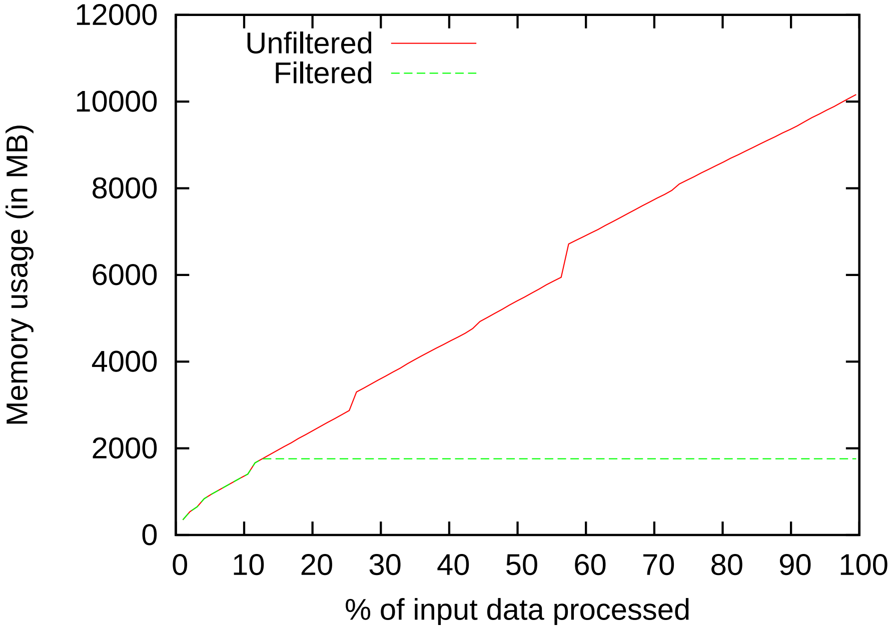

Fig. 2. Memory usage during the on-line k-mers filtering procedure, compared with un-filtered indexing. Dataset 1 is processed with parameters $m_{solid} = 10$, $r = 10$, $S_{min} = 10^7$ and using 1 thread. The first filtering pass is triggered at 11.6% of the dataset. Sporadic jumps in memory consumption correspond to resize operations of the hash table.

of nucleotides in the reads. For the *R. sphaeroides* reads set, this amounts to 0.462 GB.

We first examined the effect of on-line k-mers filtering on the first dataset. To this end, only the abundance count is retained for each k-mer. A comparison against a k-mer counting without filtering is made in Figure 2. It is important to note that, when entries corresponding to erroneous k-mers are removed from the hash table, the allocated memory is not freed but is instead made available for new entries. There are 144 M k-mers in the dataset, only 4.5 M (3.1%) of which are correct. On-line filtering enabled to keep the number of k-mers in the hash table under 23 M at any time. We verified that 4,544,973 solid kmers are retrieved without filtering, compared to 4,464,256 (98.2%) solid kmers with filtering (solid threshold $m_{solid} = 10$). The difference of 80,717 k-mers corresponds to premature filtering of k-mers that would be solid if given enough time before filtering. Then, we computed the full indexing time for an increasing number of cores (Figure 3). Some constant over-head occurs as reads pre-loading is not parallelized.

We compared memory usage of indexing procedures from other popular ultra-short reads assemblers with our implementation. The Velvet assembler (version 1.1.03) and the SOAPdenovo assembler (version 1.05) are based on de Bruijn

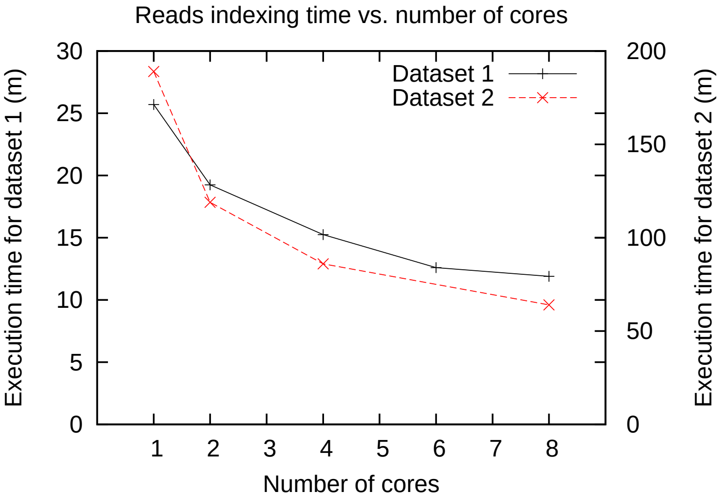

Fig. 3. Execution time of indexing for our implementation on datasets 1 and 2 using 1 node and 1 to 8 threads.

graphs and use graph simplification heuristics. SOAPdenovo is specifically optimized for memory efficiency, it discards reads and pairing information in the initial graph structure. Our implementation uses spectrum parameter $t = 4$, $S_{min} = 10^6$, $m_{solid} = 10$ and $r = 0$ for both datasets. All the assemblers are executed with k-mer size of 21. Only the indexing phase of assemblers were run (`pregraph` for SOAPdenovo, `velveth` for Velvet). Results are summarized in Table 1. The k-mers filtering step is essential in our method: complete indexing of Dataset 1 without k-mers filtering required 20.1 GB of memory. In terms of wall-clock time, these methods are comparable: for the largest dataset, SOAPdenovo and Monument completed indexing in respectively 41 and 64 minutes using 6 threads. In conclusion, our indexing scheme significantly reduces the memory bottleneck for assembly, with minor impact on parallel indexing time.

	Dataset	Monument	Velvet	SOAPdenovo
Peak memory (GB)	1	2.7	7.7	3.9
	2	15.3	-	31.4

Table 1. Practical memory usage of indexing 46 M reads *from R. sphaeroides* (dataset 1) and 320 M reads from *N.crassa* (dataset 2) using Velvet, SOAPdenovo and Monument. Velvet exceeded the memory limit (66 GB) on the second dataset.

We verified that this index permits assembly with comparable quality than existing methods. To this end, we implemented an assembly method which constructs contigs based on extensions recorded in the representative reads spectrum. We computed the following N50 values (contig length such that longer contigs produce 50% of the assembly) with respect to the same assembly size (that of Velvet, 4.30 Mbp). Our assembly of *R. sphaeroides* yields 4.28 Mbp of contigs with a N50 value of 1.49 kbp. In comparison, we executed Velvet with similar k-mer size, which yields 4.30 Mbp of contigs with a N50 value of 1.41 kbp. In terms of running times, Velvet assembly phase (`velvetg`) executed in 2 min 46 sec CPU time, whereas our parallel string graph traversal required 1 min 42 sec CPU time per thread using 6 threads.

Discussion

A novel method is proposed for multi-nodes, multi-threaded reads indexing. It introduces two filtering techniques for memory efficiency: on-line removal of erroneous k-mers and on-line indexing of only representative reads. To our knowledge, this is the first read index that provides in-memory $O(1)$ access to overlaps between reads, full read sequences and also pairing information. This novel indexing method allows to design assembly algorithms which overcome several of the performance issues inherent to graph-based assembly. For instance, memory usage is lowered as no graph structure is constructed. Independent indexing of sub-indexes allows for embarrassingly parallel and distributed computation. As implemented in our prototype, the read index is constructed using significantly less memory than recent, optimized implementations of de Bruijn graphs with comparable indexing time.

These results can also be used to reduce memory usage of single-threaded greedy assemblers. Usually, the same data structure is used to construct the final reads index and then access it during assembly. However, greedy assemblers typically uses an immutable index. Hence, taking advantage of immutability, one can focus on designing a more compact representation of the reads index once it is fully constructed. For memory efficiency, sub-indexes can be simply constructed one at a time. In our index, the hash table can be replaced by a succinct rank/select data structure [11] to represent entries, and a simple array containing fixed-length lists of representative reads. The memory overhead of such structure becomes negligible as no pointer is used. A preliminary experiment shows that the entire index of dataset 2 is represented in only 4.2 GB of memory with this method.

References

1. Ariyaratne, P.N., Sung, W.: PE-Assembler: de novo assembler using short paired-end reads. Bioinformatics (Dec 2010)
2. Boisvert, S., Laviolette, F., Corbeil, J.: Ray: Simultaneous assembly of reads from a mix of High-Throughput sequencing technologies. Journal of Computational Biology pp. 3389–3402 (2010)

3. Chapman, J.A., Ho, I., Sunkara, S., Luo, S., Schroth, G.P., Rokhsar, D.S.: Meraculous: De novo genome assembly with short Paired-End reads. PloS one 6(8), e23501 (2011)
4. Chikhi, R., Lavenier, D.: Localized genome assembly from reads to scaffolds: practical traversal of the paired string graph. Algorithms in Bioinformatics pp. 39–48 (2011)
5. Conway, T.C., Bromage, A.J.: Succinct data structures for assembling large genomes. Bioinformatics (2011)
6. Ferragina, P., Manzini, G.: Indexing compressed text. Journal of the ACM (JACM) 52(4), 552–581 (2005)
7. Jackson, B., Schnable, P., Aluru, S.: Parallel short sequence assembly of transcriptomes. BMC bioinformatics 10(Suppl 1), S14 (2009)
8. Kundeti, V., Rajasekaran, S., Dinh, H., Vaughn, M., Thapar, V.: Efficient parallel and out of core algorithms for constructing large bi-directed de bruijn graphs. BMC bioinformatics 11(1), 560 (2010)
9. Li, R., Zhu, H., Ruan, J., Qian, W., Fang, X., Shi, Z., Li, Y., Li, S., Shan, G., Kristiansen, K., Li, S., Yang, H., Wang, J., Wang, J.: De novo assembly of human genomes with massively parallel short read sequencing. Genome Research 20(2), 265–272 (2010), http://genome.cshlp.org/content/20/2/265.abstract
10. Miller, J.R., Koren, S., Sutton, G.: Assembly algorithms for next-generation sequencing data. Genomics (2010)
11. Okanohara, D., Sadakane, K.: Practical entropy-compressed rank/select dictionary. Arxiv preprint cs/0610001 (2006)
12. Peng, Y., Leung, H., Yiu, S., Chin, F., Berger, B.: IDBA - a practical iterative de bruijn graph de novo assembler. In: Research in Computational Molecular Biology, vol. 6044, pp. 426–440–440. Springer Berlin / Heidelberg (2010)
13. Raman, R., Raman, V., Satti, S.R.: Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. ACM Transactions on Algorithms (TALG) 3(4), 43–es (2007)
14. Shea, T., Williams, L., Young, S., Nusbaum, C., Jaffe, D., MacCallum, I., Przybylski, D., Gnerre, S., Burton, J., Shlyakhter, I., Gnirke, A., Malek, J., McKernan, K., Ranade, S.: ALLPATHS 2: small genomes assembled accurately and with high continuity from short paired reads. Genome Biology 10(10), R103 (2009), http://genomebiology.com/2009/10/10/R103
15. Simpson, J.T., Durbin, R.: Efficient construction of an assembly string graph using the FM-index. Bioinformatics 26(12), i367 (2010)
16. Simpson, J.T., Wong, K., Jackman, S.D., Schein, J.E., Jones, S.J., Birol: ABySS: a parallel assembler for short read sequence data. Genome research 19(6), 1117 (2009)
17. Warren, R.L., Sutton, G.G., Jones, S.J.M., Holt, R.A.: Assembling millions of short DNA sequences using SSAKE. Bioinformatics 23(4), 500–501 (2007), http://bioinformatics.oxfordjournals.org/cgi/content/abstract/23/4/500
18. Zerbino, D.R., Birney, E.: Velvet: Algorithms for de novo short read assembly using de bruijn graphs. Genome Research 18(5), 821–829 (2008), http://genome.cshlp.org/content/18/5/821.abstract