Bioinformatics, 32, 2016, i201–i208
doi: 10.1093/bioinformatics/btw279
ISMB 2016

OXFORD

Compacting de Bruijn graphs from sequencing data quickly and in low memory

Rayan Chikhi[1,*], Antoine Limasset[2] and Paul Medvedev[3,4,5]

[1]CNRS, CRIStAL, Lille, France, [2]ENS Cachan Brittany, Bruz, France, [3]Department of Computer Science and Engineering, The Pennsylvania State University, USA, [4]Department of Biochemistry and Molecular Biology, The Pennsylvania State University, USA and [5]Genome Sciences Institute of the Huck, The Pennsylvania State University, USA

*To whom correspondence should be addressed.

Abstract

Motivation: As the quantity of data per sequencing experiment increases, the challenges of fragment assembly are becoming increasingly computational. The de Bruijn graph is a widely used data structure in fragment assembly algorithms, used to represent the information from a set of reads. Compaction is an important data reduction step in most de Bruijn graph based algorithms where long simple paths are compacted into single vertices. Compaction has recently become the bottleneck in assembly pipelines, and improving its running time and memory usage is an important problem.

Results: We present an algorithm and a tool BCALM 2 for the compaction of de Bruijn graphs. BCALM 2 is a parallel algorithm that distributes the input based on a minimizer hashing technique, allowing for good balance of memory usage throughout its execution. For human sequencing data, BCALM 2 reduces the computational burden of compacting the de Bruijn graph to roughly an hour and 3 GB of memory. We also applied BCALM 2 to the 22 Gbp loblolly pine and 20 Gbp white spruce sequencing datasets. Compacted graphs were constructed from raw reads in less than 2 days and 40 GB of memory on a single machine. Hence, BCALM 2 is at least an order of magnitude more efficient than other available methods.

Availability and Implementation: Source code of BCALM 2 is freely available at: https://github.com/GATB/bcalm

Contact: rayan.chikhi@univ-lille1.fr

1 Introduction

Modern sequencing technology can generate billions of reads from a sample, whether it is RNA, genomic DNA, or a metagenome. In some applications, a reference genome can allow for the mapping of these reads; however, in many others, the goal is to reconstruct long contigs. This problem is known as fragment assembly and continues to be one of the most important challenges in bioinformatics. Fragment assembly is the central algorithmic component behind the assembly of novel genomes, detection of gene transcripts (RNA-seq) (Grabherr *et al.*, 2011), species discovery from metagenomes, structural variant calling (Iqbal *et al.*, 2012).

Continued improvement to sequencing technologies and increases to the quantity of data produced per experiment present a serious challenge to fragment assembly algorithms. For instance, while there exist many genome assemblers that can assemble bacterial sized genomes, the number of assemblers that can assemble a high-quality mammalian genome is limited, with most of them developed by large teams and requiring extensive resources (Gnerre *et al.*, 2011; Luo *et al.*, 2012; Simpson *et al.*, 2009). For even larger genomes, such as the 20 Gbp *Picea glauca* (white spruce), graph construction and compaction took 4.3 TB of memory, 38 h and 1380 CPU cores (Birol *et al.*, 2013). In another instance, the whole genome assembly of 22 Gbp *Pinus taeda* (loblolly pine) required 800 GB of memory and three months of running time on a single machine (Zimin *et al.*, 2014).

Most short-read fragment assembly algorithms use the de Bruijn graph to represent the information from a set of reads. Given a set of reads R, every distinct k-mer in R forms a vertex of the graph, while an edge connects two k-mers if they overlap by $k-1$ characters. The use of the de Bruijn graph in fragment assembly consists of a multi-step pipeline, however, the most data intensive steps are usually the first three: nodes enumeration, compaction and graph cleaning. In the first step (sometimes called k-mer counting), the set of distinct k-mers is extracted from the reads. In the second step, all unitigs (paths with all but the first vertex having in-degree 1 and all but the last vertex having out-degree 1) are compacted into a single

vertex. In the third step, artifacts due to sequencing errors and polymorphism are removed from the graph. The second and third step are sometimes alternated to further compact the graph. After these initial steps, the size of the data is reduced gradually, e.g. for a human dataset with 45× coverage,

To overcome the scalability challenges of fragment assembly of large sequencing datasets, there has been a focus on improving the resource utilization of de Bruijn graph construction. In particular, k-mer counting has seen orders of magnitude improvements in memory usage and speed. As a result, graph compaction is becoming the new bottleneck; but, it has received little attention (Kundeti *et al.*, 2010). Recently, we developed a compaction tool that uses low memory, but without an improvement in time (Chikhi *et al.*, 2014). Other parallel approaches for compaction have been proposed, as part of genome assemblers. However, most are only implemented within the context of a specific assembler, and cannot be used as modules for the construction of other fragment assemblers or for other applications of de Bruijn graphs (e.g. metagenomics).

In this paper, we present a fast and low memory algorithm for graph compaction. Our algorithm consists of three stages: careful distribution of input k-mers into buckets, parallel compaction of the buckets, and a parallel reunification step to glue together the compacted strings into unitigs. The algorithm builds upon the use of minimizers to partition the graph (Chikhi *et al.*, 2014); however, the partitioning strategy is completely novel since the strategy of Chikhi *et al.* (2014) does not lend itself to parallelization. Due to the algorithm's complexity, we formally prove its correctness. We then evaluate it on whole-genome human, pine and spruce sequencing data. The de Bruijn graph for a whole human genome dataset is compacted in roughly an hour and 3 GB of memory using 16 cores. For the >20 Gbp pine and spruce genomes, k-mer counting and graph compaction take only 2 days and 40 GB of memory, improving on previously published results by at least an order of magnitude.

2 Related work

The parallelization of de Bruijn graph compaction has been previously explored. In (Jackson *et al.*, 2010; Kundeti *et al.*, 2010), the problem is reduced to the classic list ranking problem and solved using parallel techniques such as pointer jumping. Another recurrent MPI-based approach is to implement a distributed hash table, where the k-mers and the information about their neighborhoods are distributed amongst processes. Each processor then extends seed k-mers locally as far as possible to build sub-unitigs and then passes them off to other processors for further extension. Variants of this approach are used in (Georganas *et al.*, 2014; Liu *et al.*, 2011; Simpson *et al.*, 2009). Other papers have proposed using a parallelized depth-first search (Zeng *et al.*, 2013) or a small world asynchronous parallel model (Meng *et al.*, 2014, 2012).

Before a de Bruijn graph can be compacted, it has to be constructed. Parallel approaches currently represent the state-of-the-art in this area. Many original efforts were focused on edge-centric de Bruijn graphs, where edges are represented by $(k+1)$-mers. They required the identification of both all distinct k-mers and $(k+1)$-mers (Jackson and Aluru, 2008; Jackson *et al.*, 2010; Kundeti *et al.*, 2010; Lu *et al.*, 2013; Zeng *et al.*, 2013). More recent efforts have focused on the node-centric graph, which only requires the counting of k-mers (Deorowicz *et al.*, 2014; Li *et al.*, 2013; Lu *et al.*, 2013; Marçais and Kingsford, 2011; Melsted and Pritchard, 2011; Rizk *et al.*, 2013; Simpson *et al.*, 2009).

In genome assembly, the construction and compaction of a de Bruijn graph form only the initial stages. There are also alternate approaches that do not use the de Bruijn graph at all (e.g. greedy or string graph). Numerous parallel assemblers are available for use, including ABySS (Simpson *et al.*, 2009), SOAPdenovo (Luo *et al.*, 2012), Ray (Boisvert *et al.*, 2010), PASQUAL (Liu *et al.*, 2013), PASHA (Liu *et al.*, 2011), SAND (Moretti *et al.*, 2012), SWAP-Assembler (Meng *et al.*, 2014). Other methods for parallel assembly have been published but without publicly available software (Duan *et al.*, 2014; Garg *et al.*, 2013; Georganas *et al.*, 2015; Jackson *et al.*, 2010).

There has also been work done in reducing the overall memory footprint de Bruijn graph assembly. This challenge is most pronounced for k-mer counters. However, when scaling to mammalian-sized genomes, memory usage continues to be an issue in downstream steps such as compaction. Chikhi *et al.* (2014) used minimizers to compact the de Bruijn graph of a human whole-genome dataset in under 50 MB of memory, but the algorithm did not improve the running time. Wu *et al.* (2012) propose an approach based on dividing the assembly problem into mutually independent instances. Ye *et al.* (2012) exploit the notion of graph sparseness for reducing memory use. Kleftogiannis *et al.* (2013) perform a comparative analysis and propose several memory-reducing strategies. Chikhi and Rizk (2012) use Bloom filters to reduce memory usage. Movahedi *et al.* (2012) propose a divide-and-conquer approach for compacting a de Bruijn graph.

3 Definitions

We assume, for the purposes of this paper, that all strings are over the alphabet $\Sigma = \{A, C, G, T\}$. A string of length k is called a k-mer. For a string s, we define its k-spectrum, $\mathrm{sp}^k(s)$, as the multi-set of all k-mer substrings of s. For a set of strings S, we define its multi-set k-spectrum as $\mathrm{sp}^k(S) = \cup_{s\in S}\mathrm{sp}^k(s)$. For two strings u and v, we write $u \in v$ to mean that u is a substring of v. We write $u[i..j]$ to denote the substring of u from the ith to the jth character, inclusive. We define $\mathrm{suf}_k(u) = u[|u|-k+1, |u|]$ and $\mathrm{pre}_k(u) = u[1..k]$. For two strings u and v such that $suf_k(u) = pre_k(u)$, we define a *glue* operation as $u \odot^k v = u \cdot v[k+1..|v|]$.

The binary relation $u \to v$ between two strings denotes that $suf_{k-1}(u) = pre_{k-1}(v)$. For a set of k-mers K, the *de Bruijn graph* of K is a directed graph such that the nodes are exactly the k-mers in K and the edges are given by the $\to$ relation. Note that our definition of the de Bruijn graph is node-centric, where the edges are implicit given the vertices; therefore, we use the terms de Bruijn graph and a set of k-mers interchangeably.

Suppose we are given a de Bruijn graph, represented by a set of k-mers K. Consider a path $p = (x_1, \ldots, x_m)$ over $m \geq 1$ vertices. We allow the path to be a cycle, i.e. it is possible that $x_1 = x_m$. The *endpoints* of a path are x_1 and x_m if it is not a cycle. A single-vertex path has one endpoint. A cycle does not have endpoints. The internal vertices of a path are vertices that are not endpoints. p is said to be a *unitig* if either $|p| = 1$ or for all $1 < i < m$, the out- and in-degree of x_i is 1, and the in-degree of x_m and the out-degree of x_1 are 1. A unitig is said to be maximal if it cannot be extended by a vertex on either side. The problem of compacting a de Bruijn graph is to report the set of all maximal unitigs.

We say that two strings u and v are *compactable* in a set S if $u \to v$ and, $\forall w \in S$, if $w \to v$ then $w = u$ and if $u \to w$ then $w = v$. That is, u is the only in-neighbor of v, and v is the only out-neighbor

of u. The compaction operation is defined on a pair of compactable strings and replaces u and v by a single string $u \odot^{k-1} v$.

Consider some ordering of ℓ-mers. We define the ℓ-minimizer of a string x as the smallest ℓ-mer substring of x. Given $k > \ell$ and a string x with at least k characters, we define $\text{lmm}(x)$ as the ℓ-minimizer of the prefix $(k-1)$-mer, and $\text{rmm}(x)$ as the ℓ-minimizer of the suffix $(k-1)$-mer. We refer to these as the left and right minimizers of x, respectively.

Two strings (u, v) are *m-compactable* in S if they are compactable in S and if $m = \text{rmm}(u) = \text{lmm}(v)$. The m-compaction of a set S is obtained from S by applying the compaction operation as much as possible in any order to all pairs of strings that are m-compactable in S.

4 Algorithm overview

In this section, we give a high-level description of our BCALM 2 algorithm (Algorithm 1), leaving important optimizations and implementation details to Section 6. Recall that the input is a set of k-mers K and the output are the strings corresponding to all the maximal unitigs of the de Bruijn graph of K. If time and memory are not an issue, then there is a trivial algorithm: repeatedly find compactable strings and compact them until no further compactions are possible. However, such an approach requires loading all the data into memory, which is not feasible for larger genomes.

Instead, BCALM 2 proceeds in three stages. In the first stage, the k-mers are distributed into buckets, with some k-mers being thrown into two buckets. In the second stage, each bucket is compacted, separately. In the third stage, the k-mers that were thrown into two buckets are glued back together so that duplicates are removed. Figure 1 shows the execution of BCALM 2 on a small example.

Algorithm 1. BCALM 2(K)

Input: the set of k-mers K.
1: **for all parallel** $x \in K$ **do**
2: Write x to $F(\text{lmm}(x))$.
3: **if** $\text{lmm}(x) \neq \text{rmm}(x)$ **then**
4: Write x to $F(\text{rmm}(x))$.
5: **for all parallel** $i \in \{1, \ldots, 4^\ell\}$ **do**
6: Run *CompactBucket*(i)
7: *Reunite*()

In the first stage (lines 1–6 of Algorithm 1), BCALM 2 distributes the k-mers of K to files $F(1), \ldots, F(4^\ell)$. These are called bucket files. Each k-mer $x \in K$ goes into file $F(\text{lmm}(x))$, and if $\text{lmm}(x) \neq \text{rmm}(x)$, also in $F(\text{rmm}(x))$. The parameter ℓ controls the minimizer size (in our implementation, we set $\ell = 8$).

Algorithm 2. CompactBucket(i)

1: Load $F(i)$ into memory.
2: $U \leftarrow$ i-compaction of $F(i)$.
3: **for all** strings $u \in U$ **do**
4: Mark u's prefix as "lonely" if $i \neq \text{lmm}(u)$.
5: Mark u's suffix as "lonely" if $i \neq \text{rmm}(u)$.
6: **if** u's prefix and suffix are not lonely **then**
7: Output u.
8: **else**
9: Place u in the Reunite file

In the second stage of the algorithm, we process each bucket file using the CompactBucket procedure (Algorithm 2). After the k-mer distribution of the first stage, the bucket file $F(i)$ contains all the k-mers whose left or right minimizer is i. We can therefore load $F(i)$ into memory and perform i-compaction on it. Since the size of the bucket is small, this compaction can be performed using a simple in-memory algorithm. The resulting strings are then written to disk, and will be processed during the third stage. At the end of the second stage, when all CompactBucket procedures are finished, we have performed all the necessary compactions on the data.

At this stage of the algorithm, notice that the k-mers $x \in K$ with $\text{lmm}(x) \neq \text{rmm}(x)$ exist in two copies. We call such k-mers *doubled*. We will prove in Section 5 that these k-mers are always at the ends (prefix or suffix) of the compacted strings, never internal, and they can be recognized by the fact that the minimizer at that end does not correspond to the bucket where it resides. We record these ends that have doubled k-mers by marking them 'lonely' (lines 4 and 5 of Algorithm 2), since they will need to be 'reunited' at the third stage of the algorithm. Strings that have no lonely ends are maximal unitigs, therefore they are output (line 8).

Algorithm 3. Reunite()

Input: the set of strings R from the Reunite file.
1: UF $\leftarrow$ Union find data structure whose elements are the distinct k-mer extremities in R.
2: **for all parallel** $u \in R$ **do**
3: **if** both ends of u are lonely **then**
4: $UF.union(suf_k(u), pre_k(u))$
5: **for all parallel** classes C of UF **do**
6: $P \leftarrow$ all $u \in R$ that have a lonely extremity in C
7: **while** $\exists u \in P$ that does not have a lonely prefix **do**
8: Remove u from P
9: Let $s = u$
10: **while** $\exists\, v \in P$ such that $suf_k(s) = pre_k(v)$ **do**
11: $s \leftarrow Glue(s, v)$
12: Remove v from P
13: Output s

Algorithm 4. Glue(u, v)

Input: strings u and v, such that $\text{suf}_k(u) = \text{pre}_k(v)$.
1: Let $w = u \odot^k v$.
2: Set lonely prefix bit of w to be the lonely prefix bit of u.
3: Set lonely suffix bit of w to be the lonely suffix bit of v.
4: **return** w

At the third stage of the algorithm, we process the strings output by CompactBucket with the Reunite procedure (Algorithm 3). At a high level, the purpose of Reunite is to process each string u that has a lonely end, and find a corresponding string v that has a matching lonely end with the same k-mer. When one is found, then u and v are glued together (Algorithm 4), thereby 'reuniting' the doubled k-mer that was split in the k-mer distribution stage. The new string inherits its end lonely marks from the glued strings, and the process is then repeated for the next string u that has a lonely end. After Reunite() completes, all duplicate k-mers will have been removed, and the strings in the output will correspond to the maximal unitigs.

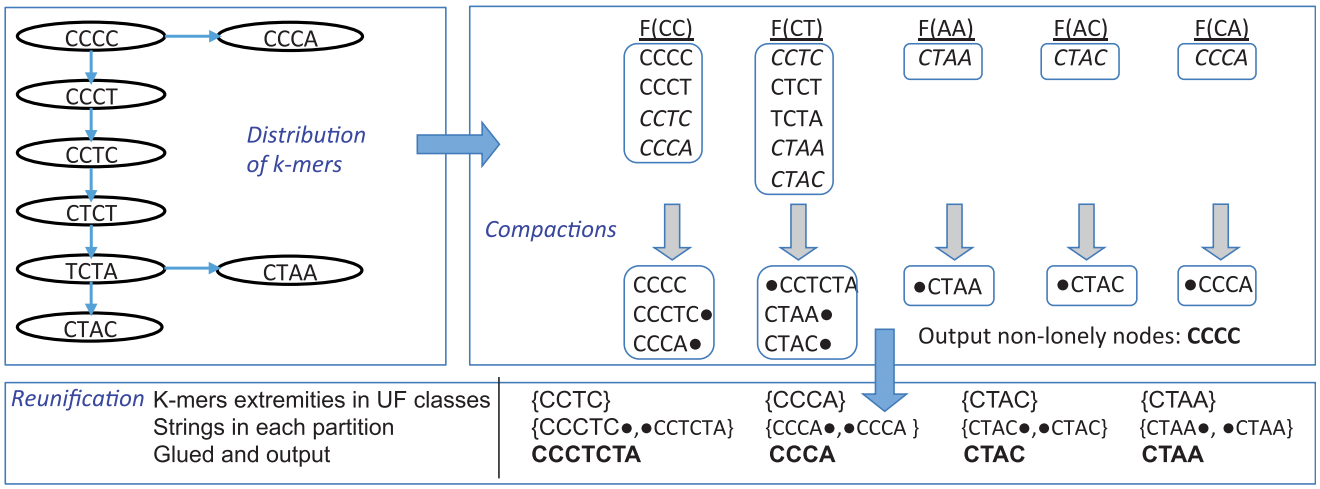

Fig. 1. Execution of BCALM 2 on a small example, with $k=4$ and $\ell=2$. On the top left, we show the input de Bruijn graph. The maximal unitigs correspond to the path from CCCT to TCTA (spelling CCCTCTA), and to the k-mers CCCC, CCCA, CTAC, CTAA. In this example, minimizers are defined using a lexicographic ordering of ℓ-mers. In the top right, we show the contents of the bucket files. Only five of the bucket files are non-empty, corresponding to the minimizers CC, CT, AA, AC and CA. The doubled k-mers are italicized. Below that, we show the set of strings that each i-compaction generates. For example in the bucket CC, the k-mers CCCT and CCTC are compacted into CCCTC, however CCCC and CCCT are not compactable because CCCA is another out-neighbor of CCCC. The lonely ends are denoted by •. In the bottom half we show the execution steps of the Reunite algorithm. Nodes in bold are output

To perform these operations efficiently in time and memory, Reunite first partitions the strings of R so that any two strings that need to be reunited are guaranteed to be in the partition. Then, each partition can be processed independently. To achieve the partition, we use a union-find (UF) data structure of all k-mers extremities. Recall that a UF data structure is created by first assigning a set to each distinct element (here, an element is the k-mer extremity of a string). Then, the union operation replaces the sets of two elements by a single set corresponding to their union. Here, union is applied to both k-mer extremities of a string. After the UF is constructed, the set of strings to be reunited is partitioned such that k-mer extremities of sequences in a partition all belong to the same UF set.

5 Proof of correctness

Recall that K is the input to the algorithm and let $\mathbb{U}$ be the strings corresponding to the set of all maximal unitigs of K. We will assume for our proof that $\mathbb{U}$ does not contain any circular unitigs. We note that since BCALM 2 outputs strings, it cannot represent circular unitigs in its output. Circular unitigs present a corner case for both the analysis and the algorithm itself, and, for the sake of presentation brevity, we do not consider them here.

We prove the correctness of BCALM 2 by showing that it outputs $\mathbb{U}$. We first give a Lemma that will allow us to show that the output is $\mathbb{U}$ by arguing about its k and $k+1$ spectrums.

LEMMA 1. Let S and T be two sets of strings of length at least k such that $\text{sp}^{k+1}(S) = \text{sp}^{k+1}(T)$ and $\text{sp}^{k}(S) = \text{sp}^{k}(T)$ and all these spectrums are without duplicates. Then, $S = T$.

PROOF. We will prove that $S \subseteq T$. The same argument will be symmetrically applicable to prove $T \subseteq S$, which will imply $S = T$.

First, we show that for all $s \in S$, there exists a $t \in T$ such that $s \in t$. Let $s \in S$ and let $p = \max\{i : \exists t \in T, s[1..i] \in t\}$, and let t be a string achieving the max. Note that $p \geq k+1$ since every $(k+1)$-mer of S is also in T. Suppose for the sake of contradiction that $p < |s|$. Then the $(k+1)$-mer $s[p-k+1, p+1]$ must occur in either another location of t or another string $t' \in T$. Either way, this means that the k-mer $s[p-k+1, p]$ must also occur elsewhere besides at $t[p-k+1, p]$. Since there are no duplicate k-mers in T, this is a contradiction.

Now, we show that $S \subseteq T$. Let $s \in S$ and let $t \in T$ such that $s \in t$. By applying an argument symmetrical to the one above, there exists a $s' \in S$ such that $t \in s'$. This means that $s \in s'$, and, in particular, $s[1..k] \in s'$. Since k-mers can only appear once in S, we must have that $s = s'$ and hence $s = t \in T$. □

Next, we characterize the k and $k+1$ spectrums of $\mathbb{U}$. Given a multi-set M, we denote by $Set(M)$ as the set version of M, with all multiplicity information implicitly removed. When referring to a set, such as K, as a multi-set, we will mean that all the elements have multiplicity one.

LEMMA 2. $\text{sp}^{k}(\mathbb{U}) = K$

PROOF. Since every vertex is a single vertex unitig path, every vertex must be covered by some maximal unitig and hence $Set(\text{sp}^{k}(\mathbb{U})) = K$. It remains to show that the set of maximal unitigs never share a vertex. First, observe that a single unitig cannot visit a vertex more than once, otherwise that vertex will be an internal vertex at one of its occurrences but will have either multiple ins or outs. We therefore need to show that no two maximal unitig paths share a vertex. Let $p = (v_1, \ldots, v_{|p|})$ and $p' = (v'_1, \ldots, v'_{|p'|})$ be two maximal unitigs that share a vertex. Because p is maximal, it cannot be a subpath of p', and cannot be a single vertex. If p is a cycle, then all its vertices have in- and out-degree one so that the only other paths it can share vertices with are sub-paths of p, contradicting the fact that p is maximal. Hence, we can assume that p and, by symmetry, p', is not a cycle.

First, suppose that all shared vertices are internal to both paths. Consider such a vertex v_i, for a maximal i. Because v_i must have different out-neighbors on both paths, it has out-degree at least two, contradicting that it is an internal vertex. Therefore there must exist at least one shared vertex that is an endpoint of one of the paths.

Suppose that v_1 is a shared vertex, and that it is not the first vertex of p', If the previous vertex of p' is not on p, then p can be extended with it, contradicting its maximality. Otherwise, consider the first vertex at which p and p' diverge. That is, the smallest $i < |p|$

such that $v_i \in p'$ but $v_{i+1} \notin p'$. The last vertex of p' must be v_i, otherwise it has out-degree at least two, contradicting that p is a unitig. Therefore, there can only exist one such vertex v_i, and it must be the last vertex of p'. □

We define a $(k+1)$-mer w as *actionable* if there exists $x \in K$ and $y \in K$ such that (x, y) are compactable in K and $w = x \odot^{k-1} y$. We define A as the multi-set of all actionable $(k+1)$-mers, but note that it does not contain duplicates because there are no duplicate k-mers in K.

LEMMA 3. $\mathrm{sp}^{k+1}(\mathbb{U}) = A$.

PROOF. First we note that neither A nor $\mathrm{sp}^{k+1}(\mathbb{U})$ have any multiple elements (by Lemma 2), and we do not need to consider multiplicities of the elements.

Suppose that there exist two k-mers x and x' such that $x \odot^{k-1} x' \in A$ but is not in $\mathrm{sp}^{k+1}(\mathbb{U})$. Because every vertex is part of some unitig, by Lemma 2 there must exist a unique unitig path $p \in \mathbb{U}$ that contains x and a unique unitig path $p' \in \mathbb{U}$ that contains x'. Note that because (x, x') are compactable, x' is the unique our-neighbor of x and x is the unique in-neighbor of x'. Also, x must be the last vertex of p and x' must be the first vertex of p'. We can therefore join p and p' by adding the edge from x to x', obtain a unitig and contradicting the maximality of p and p'.

Now suppose that there exists k-mers x and x' such that $x \odot^{k-1} x' \in \mathrm{sp}^{k+1}(\mathbb{U})$ but not in A. Let p be the unitig containing $x \odot^{k-1} x'$. Since x is not the last endpoint, it must have an out-degree of 1. Similarly, x' has an in-degree of 1. Hence, (x, x') is compactable, a contradiction. □

Next, we characterize the effect that CompactBucket() has on the k and $k+1$ spectrums. Let B be the collection of all strings u that are either output at line 7 of Compactbucket or placed in the Reunite file at line 9. We can think of these as the sum output of the CompactBucket calls.

LEMMA 4. $\mathrm{sp}^{k+1}(B) = A$ and $\mathrm{sp}^{k}(B)$ is the same as K except every doubled k-mer has multiplicity of 2 in $\mathrm{sp}^{k}(B)$.

PROOF. During distribution of the k-mers into the bucket files, every k-mer is distributed to exactly one file except for doubled k-mers, which go into two files. The compaction operations that follow do not affect the k-spectrum. Thus, the statement about $\mathrm{sp}^{k}(B)$ holds.

Initially, $\mathrm{sp}^{k+1}(K) = \varnothing$. The compaction operation changes the $k+1$ spectrum by creating one new $(k+1)$-mer. Hence, we will show that $x \odot^{k-1} y \in A$ if and only if (x, y) gets compacted at some point.

Consider an actionable $(k+1)$-mer $x \odot^{k-1} y \in A$. Observe that the right minimizer of x is the same as the left minimizer of y. Denote it by i. Because (x, y) are compactable, they are also i-compactable. The bucket file $F(i)$ will contain x and y. Because x does not have an out-neighbor that is not y in K, it will not have an out-neighbor that is not y in $F(i)$. Similarly, y will only have x as an in-neighbor in $F(i)$. Hence, (x, y) will be i-compacted in $F(i)$.

On the other hand, consider an i-compaction of $x \in K$ and $y \in K$ in $F(i)$. Any out-neighbor of x in K must have i as a left minimizer and hence must be in $F(i)$. Similarly, any in-neighbor of y in K must have i as a right minimizer and hence must also be in $F(i)$. Because (x, y) are i-compactabile in $F(i)$, x does not have an out-neighbor $y' \neq y$ in K and y does not have an in-neighbor $x' \neq x$ in K. Therefore, (x, y) are compactable in K and hence $x \odot^{k-1} y \in A$. □

Next, we analyze the third stage of the algorithm. The following two Lemmas connect the notion of loneliness to doubled k-mers.

LEMMA 5. Let $x \in K$ be a doubled k-mer. Then, x appears as a prefix of some string in R and as a suffix of some other string in R, and the ends where it appears are marked lonely.

PROOF. Let $i = \mathrm{rmm}(x)$. Since x is a doubled k-mer, $\mathrm{lmm}(x) \neq i$. Consider the fate of x in CompactBucket(i). Because CompactBucket only performs i-compactions, x will never be compacted from the left. Thus it will be a prefix of some string in U at line 2 of CompactBucket, and line 4 will mark the prefix end as lonely. The argument for the suffix is symmetrical. □

LEMMA 6. Let x be a k-mer at a lonely end of a string in R. Then, x is a doubled k-mer.

PROOF. The only way for x to be marked lonely in B would be in CompactBucket(i), for some i. Assume without loss of generality that this happens in line 4. The left minimizer of x is therefore not i, however, to have been placed into $F(i)$, its right minimizer must be i. Hence, its left and right minimizers are different and it is a doubled k-mer. □

The next Lemma is helpful to establish that each string in R that has a lonely prefix will be examined by Reunite.

LEMMA 7. Let u be a string in R with a lonely prefix. Then, there exists distinct strings $v_1, \ldots, v_\alpha$ in R such that, letting $v_0 = u, suf_k(v_i) = pre_k(v_{i-1})$ for $0 < i \leq \alpha$ and v_α has a non-lonely prefix.

PROOF. By Lemma 6, the k-mer prefix x of u is doubled, therefore by Lemma 5 there exists a string v_1 in R such that x is the suffix of v_1. If the prefix of v_1 is not lonely, then set $\alpha = 1$ and the Lemma statement is satisfied. Hence, consider the case where the prefix of v_1 is lonely.

We prove by an induction over the size of R that $v_1, \ldots, v_\alpha$ exist and satisfy the conditions stated in the Lemma. For the base case, let R be of size 2. We will prove that the prefix of v_1 is not lonely. Assume for the sake of contradiction that it is. Applying Lemmas 6 and 5 again yields that the prefix of v_1 is the suffix of another string w. Given that R is of size 2, w must be u. Hence, u and v_1 have identical k-mers extremities, they therefore spelled an isolated cycle in the input de Bruijn graph. This contradicts our assumption that $\mathbb{U}$ is free of circular unitigs, and concludes the base case.

Assume that the inductive hypothesis holds for sets of size strictly smaller than of R. Applying the hypothesis to v_1 in $R \backslash \{u\}$, there exists $v_2, \ldots, v_\alpha$ such that $suf_k(v_i) = pre_k(v_{i-1})$ for $1 < i \leq \alpha$ and v_α has a non-lonely prefix. Furthermore, $y = suf_k(v_2) = pre_k(v_1)$ and $x = suf_k(v_1) = pre_k(u)$. In addition, all strings $u, v_1, \ldots, v_\alpha$ must be distinct, else duplicates will yield circular unitigs. □

Next we analyze the effect that Reunite has on the k and $k+1$ spectrums. Let G be the final output of the algorithm.

LEMMA 8. Let $x \in K$ be a doubled k-mer. Then x appears only once in G, either internal to a string or as a non-lonely end.

PROOF. By Lemma 5, x appears as a lonely suffix of some string $u_1 \in B$ and as a lonely prefix of another string $u_2 \in B$. As a consequence of the UF data structure, u_1 and u_2 belong to the same partition P at line 6 of Algorithm 3. We will show that u_1 and u_2 are consecutively selected at line 10 of the Reunite algorithm. Observe that in Reunite, strings selected at line 10 have a lonely prefix (as a consequence of Lemma 5), and strings selected at line 7 do not.

If u_1 not does have a lonely prefix, u_1 must be selected at line 7 of Reunite. Then, u_2 is selected at the next execution of line 10. Now, assume that u_1 has a lonely prefix. Then by Lemma 7, there

exists strings $v_1, \ldots, v_\alpha$ such that $suf_k(v_i) = pre_k(v_{i-1})$ for $0 < i \leq \alpha$ and v_α has a non-lonely prefix. Then, since v_α does not have a lonely prefix, v_α is selected at line 7 of Reunite, and it follows that $v_{\alpha-1}, \ldots, v_1, u_1, u_2$ are consecutively selected at the following executions of line 10.

We conclude that $Glue(u_1, u_2)$ is performed in all cases. The action of Glue reduces the multiplicity of x from 2 to 1, and furthermore x becomes either an internal k-mer or a non-lonely end of a string in G. □

Let R_{final} be the set of strings that might remain in R at the end of the algorithm.

LEMMA 9. $\mathrm{sp}^k(G)$ has only single elements, and is equal to $Set(B)$.

PROOF. The only difference between B and $G \cup R_{\text{final}}$ is caused by executing the Glue function, which only affects the k-spectrum by changing the multiplicity of k-mer from 2 to 1. By Lemma 8, all k-mers will have multiplicity one in $G \cup R_{\text{final}}$, and hence $\mathrm{sp}^k(G \cup R_{\text{final}})$ has only single elements and is equal to $Set(B)$. It remains to show that R_{final} is empty.

All strings in R_{final} have at least one lonely end, otherwise they would have been output at line 7 of CompactBucket(). By Lemma 6, such a lonely end must be a doubled k-mer. However, by Lemma 8, all doubled k-mers are either internal or non-lonely ends in G. Therefore, R_{final} must be empty. □

Finally, we are ready to prove the correctness of BCALM 2.

THEOREM 1. BCALM 2 outputs $\mathbb{U}$.

PROOF. We will show that the conditions of Lemma 1 are satisfied for G and $\mathbb{U}$. The glue operation does not change the $(k+1)$-spectrum, and $R_{final} = \emptyset$, so $\mathrm{sp}^{k+1}(B) = \mathrm{sp}^{k+1}(G \cup R_{final}) = \mathrm{sp}^{k+1}(G)$. Combining this with Lemma 3 and Lemma 4, we get that $\mathrm{sp}^{k+1}(G) = \mathrm{sp}^{k+1}(B) = A = \mathrm{sp}^{k+1}(\mathbb{U})$ and that, because A is duplicate free by definition, these spectrums do not contain duplicates. Combining Lemma 4 and Lemma 9, we also get that $\mathrm{sp}^k(G) = K$ and by Lemma 2, $\mathrm{sp}^k(\mathbb{U}) = K$. □

6 Optimizations and implementation

In this section, we describe some of the optimizations and important implementation details that we used to implement the pseudocode of Section 4.

For the sake of brevity, we have only described the algorithm for the directed de Bruijn graph. In our implementation, we extend the algorithm to the bidirected graph model (Kececioglu, 1992; Medvedev *et al.*, 2007), in the natural way, to handle the double-stranded nature of DNA.

To compute minimizers, we do not use a lexicographical ordering of ℓ-mers, as this has been previously shown to lead to unbalanced bucket files and increased memory usage (Chikhi *et al.*, 2014; Deorowicz *et al.*, 2014). Deorowicz *et al.* (2014) proposed to use the lexicographic order but to forbid certain well known frequent ℓ-mers from being minimizers (e.g. the poly-A). We use frequency based minimizers, which we proposed in an earlier work (Chikhi *et al.*, 2014). In this approach, an initial ℓ-mer counting step is performed on the data and ℓ-mers are ordered by increasing frequency. Because ℓ is small, the time and memory for this step is negligible.

Buckets are organized into groups, in order to introduce natural checkpoints in BCALM 2 in between parallel sections. BCALM 2 iterates sequentially through the groups, but parallelizes the processing within a group. The For loop at line 1 of Algorithm 1 is executed in parallel within a group, with each thread given a subset of K. k-mers are distributed only to those buckets that are in the group, with other buckets being ignored. Bucket files are implemented as thread-safe queues, as opposed to physical files on disk. The statements at lines 2 and 4 of Algorithm 1 enqueue x into the appropriate queue, and Algorithm 2 dequeues them at line 1, instead of reading them from disk. After the k-mers are distributed, buckets from a group are compacted in parallel. The CompactBucket routines are independent of each other, and hence we run $CompactBucket(i)$ in parallel using all available processors. After a BCALM 2 finishes processing a group, it moves on to the next group.

To reduce memory of the UF data structure, we created a minimal perfect hash function (MPHF) (Cormen, 2009) of all distinct k-mer extremities in the Reunite file (denote their number as d). The UF structure is therefore implemented as a vector v of MPHF indices, of total size $d \log d$. The UF class of a given k-mer is therefore $v[x]$, where x is the MPHF index of the k-mer.

The BCALM 2 algorithm takes as input a set of distinct k-mers. However, in our implementation, BCALM 2 is developed using the GATB library (Drezen *et al.*, 2014), allowing it to seamlessly integrate GATB's k-mer counter. Therefore, the BCALM 2 software takes reads as input, and executes this k-mer counter prior to compaction. This is a disk-based algorithm inspired by KMC2 (Deorowicz *et al.*, 2014) and DSK (Rizk *et al.*, 2013). In this k-mer counter, k-mers are divided into partitions according to their minimizer, then each partition is counted independently. We modified the GATB k-mer counting algorithm so that partition files correspond exactly to bucket groups. We obtained further optimizations by representing strings using two bits per character.

7 Experimental results

We evaluated the scalability of BCALM 2, and how it compares to other tools for compacting the de Bruijn graph. Experiments were run on a single machine equipped with an Intel Xeon CPU with 32 cores clocked at 2.76 GHz and 512 GB of memory. We used two human sequencing datasets from the GAGE benchmark (Salzberg *et al.*, 2011) and from and two larger datasets from the spruce and pine sequencing projects (Birol *et al.*, 2013; Zimin *et al.*, 2014).

7.1 Human datasets

The first dataset is Illumina reads from a human chromosome 14 (36 million, 155 bp each, 2.9 GB compressed FASTQ). The second dataset is Illumina reads from the whole human genome NA18507 (1.4 billion, 100 bp each, 54 GB compressed FASTQ, SRA SRX016231).

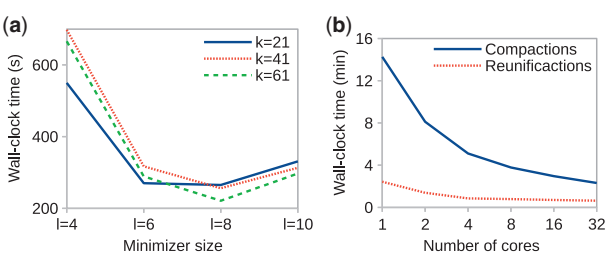

Fig. 2. BCALM 2 wall-clock running times with respect to (**a**) parameters ℓ and k (using 4 cores) and (**b**) number of cores (using $k = 55$ and $\ell = 8$), on the chromosome 14 dataset

We first evaluate how BCALM 2 is affected by changes in the parameters k (k-mer size) and ℓ (minimizer length). Figure 2(a) shows that BCALM 2 has nearly identical running times for $6 \leq \ell \leq 10$, and across all tested k values. Shorter minimizers sizes such as $\ell = 4$ create fewer buckets, hence limit parallel speedups. Second, we evaluate how well BCALM 2 scales with multiple processors. Figure 2(b) shows that compaction and Reunite steps scale almost linearly with the number of threads. There remains overheads related to disk I/O.

We compare the performance of BCALM 2 to other available implementation of compaction algorithms: (i) our own previous serial compaction algorithm BCALM (Chikhi *et al.*, 2014), (ii) the parallel ABYSS-P step of the ABySS assembler (version 1.9.0), excluding bubble removal (Simpson *et al.*, 2009), (iii) the parallel compaction step of the Meraculous 2 assembler (version 2.0.5), executed from the *mergraph* to the *contigs* step (Georganas *et al.*, 2014) and (iv) the single-threaded unitig construction step of the Minia assembler (version 2.0.3) (Chikhi and Rizk, 2012). There are other promising stand-alone tools that implement parallel de Bruijn graph compaction, but we found them to either not be publicly available (Jackson *et al.*, 2010) or unable to run on real mammalian data because of an upper bound of 31 on the k-mer size (Liu *et al.*, 2011; Meng *et al.*, 2014). For BCALM, the datasets were first processed using the DSK k-mer counting software (Rizk *et al.*, 2013) to generate the set of k-mers.

In addition to the results shown in Table 1, Minia took 27 h and 7 GB of memory on the whole human dataset (using identical k and abundance cutoff as in the table). For ABySS-P, the shown numbers include the k-mer counting step, which could not be extricated from the software. For the purposes of comparison, the k-mer counting step to generate the input for BCALM 2 completed in 46 mins and 2 GB of memory for the whole human dataset.

Table 1 shows that BCALM 2 outperforms existing techniques in terms of running time. Since multiple graph compactions are done in parallel, BCALM 2 requires more memory than BCALM, however it is more memory-efficient than Meraculous 2.

Table 1. Running times (wall-clock) and memory usage of compaction algorithms for the human datasets.

Dataset	BCALM 2	BCALM	ABySS-P	Meraculous 2
Chr 14	5 mins	15 mins	11 mins	62 mins
	400 MB	19 MB	11 GB	2.35 GB
Whole human	1.2 h	12 h	6.5 h	16 h*
	2.8 GB	43 MB	89 GB	unreported*

For BCALM 2 and BCALM we used $k = 55$, and $\ell = 8$ and $\ell = 10$, respectively; abundance cutoffs were set to 5 for Chr 14 and 3 for whole human. We used 16 cores for the parallel algorithms ABySS, Meraculous 2 and BCALM 2. Meraculous 2 aborted with a validation failure due to insufficient peak k-mer depth when we ran it with abundance cutoffs of 5. We were able to execute it on chromosome 14 with a cutoff of 8, but not for the whole genome. (*)For the whole genome, we show the running times given in Georganas et al. (2014). The exact memory usage was unreported there but is less than <1 TB. Meraculous 2 was executed with 32 prefix blocks.

Table 2. Performance of BCALM 2 on the loblolly pine and white spruce datasets.

Dataset	Loblolly pine	White spruce
Distinct k-mers ($\times 10^9$)	10.7	13.0
Num threads	8	16
CompactBucket() time	4 h 40 m	3 h 47 m
CompactBucket() mem	6.5 GB	6 GB
Reunite file size	85 GB	140 GB
Reunite() time	4 h 32 m	3 h 08 m
Reunite() memory	31 GB	39 GB
Total time	9 h 12 m	6 h 55 m
Total max memory	31 GB	39 GB
Unitigs ($\times 10^6$)	721	1200
Total length	32.3 Gbp	49.0 Gbp
Longest unitig	11.2 kbp	9.0 kbp

The k-mer size was 31 and the abundance cutoff for k-mer counting was 7.

7.2 Pine and spruce datasets

We further evaluated BCALM 2 on two very large sequencing datasets: Illumina reads from the 20 Gbp *Picea glauca* genome (8.5 billion reads, 152–300 bp each, 1.1 TB compressed FASTQ, SRA056234) (Birol *et al.*, 2013), and Illumina paired-end reads from the 22 Gbp *Pinus taeda* genome (9.4 billion reads, 128–154 bp each, 1.2 TB compressed FASTQ, SRX016231). The k-mer counting step took around a day and <40 GB of memory for each dataset.

Table 2 shows the performance of BCALM 2 on these two datasets, as well as unitigs statistics. Graph construction of the spruce dataset previously required 4.3 TB of memory and 2 days on a 1380-core cluster (Birol *et al.*, 2013), while the assembly of the pine dataset previously required 800 GB of memory and 3 months on a single machine (Zimin *et al.*, 2014). Another execution of BCALM2 on the same datasets using a value of k=61 shows similar performance, see Supplemental Table 1.

Although we used the same sequencing datasets, several parameters differ between these previous reports and our results (e.g. k value, abundance cutoff, and whether reads were error-corrected). Hence run time, memory usage, and unitigs statistics cannot be directly compared. However, it seems reasonable to infer that BCALM 2 would remain 1–2 orders of magnitude more efficient in time and memory.

In addition, we tested the robustness of BCALM 2 to an even larger number of erroneous k-mers by reducing the k-mer abundance cutoff to 2. The k-mer counting and compactions steps completed also within 2 days and 40 GB of memory. The resulting unitig file was much larger (resp. 67 GB and 107 GB). This is expected, due to a large number of sequencing errors resulting in erroneous k-mers being incorporated into the graph (roughly 2 billion k-mers in both cases, i.e. $\approx 2k \times 10^9 = 62$ Gbp of new unitigs). A non-negligible amount of sequencing errors is also likely present in the data presented in Table 2.

8 Discussion

In this paper, we present BCALM 2, an open-source parallel and low-memory tool for the compaction of de Bruijn graphs. BCALM 2 constructed the compacted de Bruijn graph of a human genome sequencing dataset in 76 mins and 3 GB of memory. Furthermore, k-mer counting and graph compaction using BCALM 2 of the 20 Gbp white spruce and the 22 Gbp loblolly pine sequencing datasets required only 2 days and 40 GB of memory each.

BCALM 2 is different from previous approaches in several regards. First, it is a separate module for compaction, with the goal that it can be used as part of any other tools that build the de Bruijn graph. While parallel genome assemblers offer impressive performance, there are many situations where differences in data require the development of a new assembler, and hence it is desirable to build

modular components. Second, we do not aim at a method that can be distributed on a cluster over thousands of nodes. While clearly powerful, such machines are not usually accessible to a biology lab, and we believe that a tool that uses a shared memory multi-core machine is more applicable. Methods that are designed for multi-node clusters will often consume a prohibitive amount of memory when run on multiple threads of a shared memory machine.

Acknowledgements

We would like to thank Kamesh Madduri for helpful discussions, Colleen O'Rourke and Guillaume Rizk for code contributions. We used computing resources from the ICS@PSU and GenOuest infrastructures.

Funding

This work has been supported in part by NSF awards DBI-1356529, CCF-1439057, IIS-1453527 and IIS-1421908 to PM.

Conflict of Interest: none declared.

References

Birol,I. *et al*. (2013) Assembling the 20 gb white spruce (*Picea glauca*) genome from whole-genome shotgun sequencing data. *Bioinformatics*. **29**, 1492–1497.

Boisvert,S. *et al*. (2010) Ray: simultaneous assembly of reads from a mix of high-throughput sequencing technologies. *J. Comput. Biol.*, **17**, 1519–1533.

Chikhi,R. and Rizk,G. (2012) Space-efficient and exact de Bruijn graph representation based on a Bloom filter. In: *WABI*, volume **7534** of *LNCS*, pp. 236–248. Springer.

Chikhi,R. *et al*. (2014) On the representation of de Bruijn graphs. *In: Research in Computational Molecular Biology*. Springer, Berlin, pp. 35–55.

Cormen,T.H. (2009) *Introduction to Algorithms*. MIT Press, Cambridge.

Deorowicz,S. *et al*. (2014) Kmc 2: Fast and resource-frugal *k*-mer counting. *arXiv Preprint arXiv*, 1407.1507.

Drezen,E. *et al*. (2014) GATB: genome assembly & analysis tool box. *Bioinformatics*, **30**, 2959–2961.

Duan,X. *et al*. (2014). Hipga: A high performance genome assembler for short read sequence data. In: *IEEE IPDPSW 2014*. IEEE, pp. 576–584.

Garg,A. *et al*. (2013). Ggake: Gpu based genome assembly using k-mer extension. In: *IEEE HPCC_EUC 2013*. IEEE, pp. 1105–1112.

Georganas,E. *et al*. (2014). Parallel de bruijn graph construction and traversal for de novo genome assembly. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. IEEE Press, pp. 437–448.

Georganas,E. *et al*. (2015). Hipmer: an extreme-scale de novo genome assembler. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, pp. 14.

Gnerre,S. *et al*. (2011) High-quality draft assemblies of mammalian genomes from massively parallel sequence data. *PNAS*, **108**, 1513.

Grabherr,M.G. *et al*. (2011) Full-length transcriptome assembly from RNA-Seq data without a reference genome. *Nat. Biotechnol.*, **29**, 644–652.

Iqbal,Z. *et al*. (2012) De novo assembly and genotyping of variants using colored de Bruijn graphs. *Nat. Genet.*, **44**, 226–232.

Jackson,B.G. and Aluru,S. (2008) Parallel construction of bidirected string graphs for genome assembly. In: ICPP'08. 37th International Conference on Parallel Processing, *2008*. IEEE, pp. 346–353.

Jackson,B.G. *et al*. (2010). Parallel de novo assembly of large genomes from high-throughput short reads. In: *IEEE IPDPS 2010*. IEEE, pp. 1–10.

Kececioglu,J.D. (1992) *Exact and Approximation Algorithms for DNA Sequence Reconstruction*. Ph.D. thesis, University of Arizona, Tucson, AZ, USA.

Kleftogiannis,D. *et al*. (2013) Comparing memory-efficient genome assemblers on stand-alone and cloud infrastructures. *PloS One*, **8**, e75505.

Kundeti,V.K. *et al*. (2010) Efficient parallel and out of core algorithms for constructing large bi-directed de Bruijn graphs. *BMC Bioinformatics*, **11**, 560.

Li,Y. *et al*. (2013) Memory efficient minimum substring partitioning. *Proc. VLDB Endowment*, **6**, 169–180.

Liu,X. *et al*. (2013) Pasqual: parallel techniques for next generation genome sequence assembly. *IEEE Trans. Parallel Distributed Syst.*, **24**, 977–986.

Liu,Y. *et al*. (2011) Parallelized short read assembly of large genomes using de bruijn graphs. *BMC Bioinformatics*, **12**, 354.

Lu,M. *et al*. (2013) Gpu-accelerated bidirected de bruijn graph construction for genome assembly. In: *Web Technologies and Applications*. Springer, Berlin, pp. 51–62.

Luo,R. *et al*. (2012) SOAPdenovo2: an empirically improved memory-efficient short-read de novo assembler. *GigaScience*, **1**, 18.

Marçais,G. and Kingsford,C. (2011) A fast, lock-free approach for efficient parallel counting of occurrences of k-mers. *Bioinformatics*, **27**, 764–770.

Medvedev,P. *et al*. (2007) Computability of models for sequence assembly. *WABI*, 289–301.

Melsted,P. and Pritchard,J.K. (2011) Efficient counting of k-mers in DNA sequences using a Bloom filter. *BMC Bioinformatics*, **12**, 333.

Meng,J. *et al*. (2012) Small world asynchronous parallel model for genome assembly. In: *Network and Parallel Computing*. Springer, Berlin, pp. 145–155.

Meng,J. *et al*. (2014). SWAP-Assembler: Scalable and efficient genome assembly towards thousands of cores. In: *RECOMB-Seq 2014*.

Moretti,C. *et al*. (2012) A framework for scalable genome assembly on clusters, clouds, and grids. *IEEE Trans. Parallel Distributed Syst.*,, **23**, 2189–2197.

Movahedi,N.S. *et al*. (2012) De novo co-assembly of bacterial genomes from multiple single cells. In: *IEEE BIBM 2012*. IEEE, pp. 1–5.

Rizk,G. *et al*. (2013) DSK: k-mer counting with very low memory usage. *Bioinformatics*, **29**, 652–653.

Salzberg,S.L. *et al*. (2011) GAGE: a critical evaluation of genome assemblies and assembly algorithms. *Genome Res*, **22**, 557–567.

Simpson,J.T. *et al*. (2009) ABySS: a parallel assembler for short read sequence data. *Genome Res.*, **19**, 1117–1123.

Wu,X.L. *et al*. (2012) Tiger: tiled iterative genome assembler. *BMC Bioinformatics*, **13**, S18.

Ye,C. *et al*. (2012) Exploiting sparseness in de novo genome assembly. *BMC Bioinformatics*, **13**, S1.

Zeng,L. *et al*. (2013) Improved parallel processing of massive de bruijn graph for genome assembly. In: *Web Technologies and Applications*. Springer, Berlin, pp. 96–107.

Zimin,A. *et al*. (2014) Sequencing and assembly of the 22-gb loblolly pine genome. *Genetics*, **196**, 875–890.