BIOINFORMATICS

Vol. 00 no. 00 2012
Pages 1–9

An Efficient Partitioning Algorithm for *De Novo* Assembly

Yang Li, Xifeng Yan*

Department of Computer Science, University of California at Santa Barbara

Received on XXXXX; revised on XXXXX; accepted on XXXXX

Associate Editor: XXXXXXX

ABSTRACT

Motivation: A major challenge in next-generation genome sequencing (NGS) is to assemble massive overlapping short reads that are randomly sampled from DNA fragments. The length of short reads ranges from a few tens to several hundreds of bases. To complete assembling, one needs to solve two fundamental tasks in many leading assembly algorithms: (1) Count the number of occurrences of k-mers (length-k substrings in DNA sequences) to perform error correction, and (2) Build a de Bruijn graph from short reads and then traverse the graph to finish assembling. For large genomes, both tasks consume a huge amount of main memory, making it impossible for large-scale parallel assembly of genome sequences on commodity servers.

Results: In this paper, we develop Minimum Substring Partitioning (MSP), a disk-based partitioning approach, to efficiently perform k-mer counting and construct de Bruijn graphs for large genomes using a small amount of memory. MSP breaks short reads into multiple disjoint partitions such that each partition can be loaded into memory, processed individually and later merged with others to finish the tasks. By making full use of the overlaps among the k-mers derived from the same short read, MSP can achieve astonishing compression ratio so that the I/O cost can be significantly reduced. For the task of k-mer counting, our method offers a very fast and memory-efficient solution. For NGS assembly, it uses small memory footprint in building de Bruijn graph. Our approach is general; it can be applied to any de Bruijn graph based assembly algorithm.

Availability: The MSP program for k-mer counting and de Bruijn graph construction is available at http://www.cs.ucsb.edu/∼xyan/MSP

Contact: xyan@cs.ucsb.edu

1 INTRODUCTION

High-quality genome sequencing plays an important role in genome research. A central problem in genome sequencing is assembling massive short reads generated by the next-generation sequencing technologies (Mardis *et al*., 2008). These reads are usually randomly extracted from samples of DNA segments. Typically a modern technology can produce billions of short reads whose length varies from a few tens of bases to several hundreds. For example, massively parallel sequencing platforms, such as Illumina (www.illumina.com), SOLiD (www.appliedbiosystems.com), and 454 Life Sciences (Roche) GS FLX (www.roche.com), can produce reads from 25 to 500 bases in length. The short read length is expected to further increase in the following years.

Despite the progress in sequencing techniques and assembly methods in recent years, de novo assembly remains a computationally challenging task. The existing de novo assembly algorithms can be classified into two main categories based on their internal assembly model: (1) The overlap-layout-consensus model, used by Celera (Myers *et al*., 2000), ARACHNE (Batzoglou *et al*., 2002), Atlas (Havlak *et al*., 2004), Phusion (Mullikin *et al*., 2003) and Forge (Platt *et al*., 2010); (2) The de Bruijn graph model, used by Euler (Pevzner *et al*., 2001), Velvet (Zerbino *et al*., 2008), ABySS (Simpson *et al*., 2009), AllPaths (Butler *et al*., 2008) and SOAPdenovo (Li *et al*., 2010a). The overlap-layout-consensus model builds an overlap graph between reads. Since each read can overlap with many other reads, it is more useful for sequencing data sets with a small number of long reads. The de Bruijn graph approach breaks short reads to k-mers (substring of length k) and then connects k-mers according to their overlap relations in the reads. The de Bruijn graph approach is usually able to assemble larger quantities (e.g., billions) of short reads with greater coverage. Systematic comparison of these algorithms is given by Earl *et al*. (2011) and Salzberg *et al*. (2012).

Although the de Bruijn graph approach comes up with a good framework to reduce the computation time for assembly, the graph size can be extremely large, for example, containing billions of nodes (k-mers) for genomes of higher eukaryotes like mammals. Therefore, large memory consumption is a pressing practical problem for the de Bruijn graph based approach (Miller *et al*., 2010). For the short read sequences generated from mammalian-sized genomes, the single machine software like Euler, Velvet, AllPaths and SOAPdenovo will not be able to finish assembling successfully within a reasonable amount of memory. Due to this drawback, it significantly limits the opportunity to run de novo assembly on numerous commodity machines in parallel for large-scale sequence analysis. This problem has also blocked other application of de Bruijn graphs, e.g., variants discovery in Iqbal *et al*. (2012). In this work, we are going to explore a memory-efficient de Bruijn graph construction algorithm.

To deal with the memory issue, several methods were proposed to store k-mers efficiently. Most NGS assemblers encode each nucleotide, A, C, G, and T using 2 bits, so that each k-mer occupies $\lceil k/4 \rceil$ bytes. These k-mers are then stored in a hash table to merge duplicate k-mers. For example, in the Asian genome short read data set (Li *et al*., 2010a), if $k = 25$, there are about 14.6 billion distinct k-mers. Assuming a load factor of $2/3$ for the hash table, the k-mers would require nearly 160 GB memory. When k increases, this number will be larger and beyond the memory capacity of commodity servers. Therefore, distributed assembly software was developed, e.g., ABySS (MPI-based) (Simpson *et al*., 2009). ABySS uses a

*to whom correspondence should be addressed

hash function to distribute k-mers evenly across a cluster so that the memory requirement for each machine is not that big. However, ABySS suffers from a big performance problem: it cannot ensure adjacent k-mers being hashed to the same machine. Therefore these adjacent k-mers located at different machines would cause intensive cross-machine communication during the assembly process. The disk-based partitioning technique proposed in this work can get rid of this problem: it not only generates small partitions, which only require several gigabytes memory to process, but also retains the majority of the adjacent k-mers in the same partition, which will greatly reduce the cross-machine communication and facilitate distributed parallel processing.

Before constructing de Bruijn graphs, an error correction step is often taken to eliminate erroneous k-mers. In most NGS data sets, a large fraction of k-mers arise from sequencing errors. These k-mers have very low frequencies. In the giant panda genome sequencing experiment (Li *et al*., 2010b), the error correction process could eliminate 68% of the observed 27-mers, reducing the total number of distinct 27-mers from 8.62 billion to 2.69 billion. Though error correction is usually helpful, obtaining the k-mer frequencies itself is a computationally demanding task for large genome data sets. One solution is using a hash table, where keys are the k-mers and values are the corresponding k-mer frequencies. Unfortunately, this approach will blow up main memory, as we can see from the example in the last paragraph. Furthermore, this problem will become severe when the length of short reads and k-mers produced by the next-generation sequencing techniques further increases.

A recently developed program called Jellyfish (Marcais *et al*., 2011) is designed to count k-mers in a memory efficient way. It adopts a "quotienting" technique to reduce the memory consumption of k-mers stored in a hash table. Implemented with a multi-threaded, lock-free hash table, it is able to count k-mers up to 31 nucleotides in length using a much smaller amount of memory than the previous methods. When there is no enough memory to carry out the entire computation, Jellyfish will write intermediary counting results to disk and later merge them. Since the same k-mer may appear in several different intermediary results, the merge operation is not just a simple concatenation process; it can be quite slow. Another state-of-the-art k-mer counting algorithm, BFCounter (Melsted *et al*., 2011) is based on bloom filter, a probabilistic data structure that can also reduce memory footprint. However, BFCounter is 3 times slower than Jellyfish when Jellyfish is able to finish the task in memory (Melsted *et al*., 2011). And moreover, it might miss some counts.

Figure 1 shows a general pipeline of a de Bruijn graph based assembly process. The most memory intensive and time consuming parts are the step to count k-mers along with error correction and the step to build de Bruijn graph. For example, in an assembly experiment conducted on the Human African sequencing data set (Li *et al*., 2010a), the error correction step (which depends on k-mer counting) used 96 GB memory along with 55% of the total time; and the de Bruijn graph construction step used 140 GB memory along with 20% of the total time; while the other steps used no more than 62 GB memory. Therefore error correction and de Bruijn graph construction are the two main computational bottlenecks in de Bruijn graph based assembly methods.

In this paper, we propose Minimum Substring Partitioning (MSP), a disk-based approach, to efficiently perform k-mer counting and construct de Bruijn graph for large genomes using very small

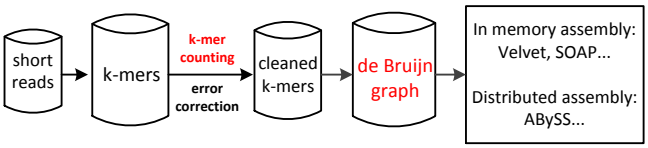

Fig. 1. de Bruijn graph based sequence assembly pipeline

amount of memory. MSP breaks short reads to "super k-mers" (substring of length greater than or equal to k) such that each "super k-mer" contains k-mers sharing the same minimum p-substring ($p \leq k$). The effect is equivalent to compressing consecutive k-mers using the original sequences. It is shown that this compression approach does not introduce significant computing overhead, but could lead to partitions 10-15 times smaller than the direct approach using a hash function, thus greatly reducing I/O cost.

We successfully applied MSP to k-mer counting and de Bruijn graph construction. Experimental results show that our counting method is faster and more memory efficient than both Jellyfish and BFCounter; and our graph construction method can build de Bruijn graphs efficiently in one machine using a reasonable amount of memory on data sets that are beyond the processing capability of existing single machine de novo assembly algorithms. MSP can be widely used by applications that need to build a de Bruijn graph from a large amount of short reads. Its concept is also valuable to distributed de novo assembly.

2 BACKGROUND

DEFINITION 1 (Short Read, K-Mer). *A short read is a string over alphabet $\Sigma = \{A, C, G, T\}$ (in DNA assembly). A k-mer is a string over Σ whose length is k. Given a short read s, $s[i, j]$ denotes the substring of s from the i_{th} element to the j_{th} element (both inclusive). s can be broken into $n - k + 1$ k-mers, written as $s[1, k]$, $s[2, k + 1], \ldots, s[n - k + 1, n]$. Two k-mers in s, $s[i, k + i - 1]$, $s[i + 1, k + i]$ are called adjacent in s.*

We can view k-mers generated in a way that a window with width k slides through a short read s. The adjacency relationship exists between each pair of k-mers for which the last k-1 bases of the first k-mer are exactly the same as the first k-1 bases of the last k-mer. .

DEFINITION 2 (Reverse Complement). *DNA sequences can be read in two directions: forwards and backwards with each nucleotide changed to its Watson-Crick complement ($A \leftrightarrow T$ and $C \leftrightarrow G$). For each DNA sequence, its corresponding read in the other direction is called reverse complement and they are considered equivalent in bioinformatics.*

In most sequencing technologies, the fragments (short reads) are randomly extracted from the DNA sequence in either direction. Therefore, if two k-mers, K_1 and K_2, are adjacent from K_1 to K_2 in the short reads data set, it implies that the reverse complement k-mer of K_2, say K_2' and the reverse complement k-mer of K_1, say K_1', are adjacent from K_2' to K_1'. So in an assembly processing, each short read should be read twice, once in forward direction and then in the reverse complement direction. However, in real implementation, it is possible to avoid reading sequences twice by inferring the

subgraph introduced by reverse complements later from the forward direction subgraph.

DEFINITION 3 (De Bruijn Graph). *Given a short read set $S = \{s_i\}$, a de Bruijn graph $G = \{V, E\}$ is constructed by creating a vertex for every distinct k-mer in S and connecting two vertices with a directed edge if their corresponding k-mers are adjacent in at least one short read.*

It is clear that each vertex in a de Bruijn graph built from DNA short reads can have at most $|\Sigma| = 4$ direct predecessor vertices and $|\Sigma| = 4$ direct successor vertices. In a de Bruijn graph, a pair of adjacent k-mers are connected by an edge if and only if they appear in the same short read. It is possible that the same edge may appear in multiple short reads. The number of times that an edge (a (k+1)-substring) is observed in the short reads is called the multiplicity of that edge. Figure 2 shows a de Bruijn graph generated from two short reads with k being 3. The edge weight shows the multiplicity of the edge. For simplicity reason, we do not depict the k-mers generated by the reverse complements of short reads here.

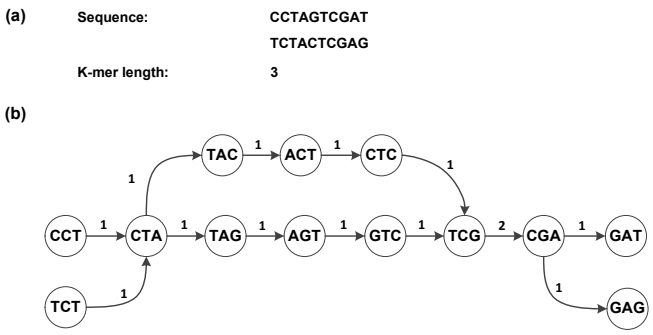

Fig. 2. A de bruijn graph example: k=3 (reverse complements omitted)

3 MINIMUM SUBSTRING PARTITIONING

Our approach to do fast and memory efficient k-mer counting and de Bruijn graph construction is based on a disk-based partition approach called Minimum Substring Partitioning (MSP). MSP is able to partition k-mers into multiple disjoint partitions, as well as retaining adjacent k-mers in the same partition. This nice property introduces two advantages: first, instead of being outputted as several individual k-mers, consecutive k-mers can be compressed to "super k-mers" (substring of length greater than or equal to k), which will greatly reduce the I/O cost of partitioning; second, with adjacent k-mers in the same partition, it is possible to build local graph for each partition in parallel and later merge them to generate the global de Bruijn graph.

DEFINITION 4 (Substring). *A substring of a string $s = s_1 s_2 \dots s_n$ is a string $t = s_{i+1} s_{i+2} \dots s_{i+m}$, where $0 \leq i$ and $i + m \leq n$.*

DEFINITION 5 (Minimum Substring). *Given a string s, a length-p substring t of s is called the minimum p-substring of s, if $\forall s'$, s' is a length-p substring of s, s.t., $t \leq s'$ ($\leq$ defined by lexicographical order). The minimum p-substring of s is written as $min_p(s)$.*

DEFINITION 6 (Minimum Substring Partitioning). *Given a string $s = s_1 s_2 \dots s_n$, $p, k \in N$, $p \leq k \leq n$, minimum substring partitioning breaks s to substrings with maximum length $\{s[i,j] | i + k - 1 \leq j, 1 \leq i, j \leq n\}$, s.t., all k-mers in $s[i,j]$ share the same minimum p-substring, and it is not true for $s[i, j+1]$ and $s[i-1, j]$. $s[i,j]$ is also called "super k-mer".*

Minimum Substring Partitioning comes from the intuition that two adjacent k-mers are very likely to share the same minimum p-substring if $p << k$, since there is a length-(k-1) overlap between them. Figure 3 shows a Minimum Substring Partitioning example. In this example, the first 4 k-mers have the same minimum 4-substring, $ACAC$, as highlighted in red box; and the last 3 k-mers share the same minimum 4-substring, $ACCC$, as highlighted in blue box. In this case, instead of generating all these 7 k-mers separately, we can just compress them using the original short read. Namely, we compress the first 4 k-mers to $CTGACACTTGACCCGTGGT$, and output it to the partition corresponding to the minimum 4-substring $ACAC$. Similarly, the last 3 k-mers are compressed to $CACTTGACCCGTGGTCAT$ and outputted to the partition corresponding to the minimum 4-substring $ACCC$. Generally speaking, given a short read $s = s_1 s_2 \dots s_n$, if the adjacent j k-mers from $s[i, i+k-1]$ to $s[i+j-1, i+j+k-2]$ share the same minimum p-substring t, then we can just output substring $s_i s_{i+1} \dots s_{i+j+k-2}$ to the partition corresponding to the minimum p-substring t without breaking it to j individual k-mers. If j is large, this compression strategy will dramatically reduce the I/O cost.

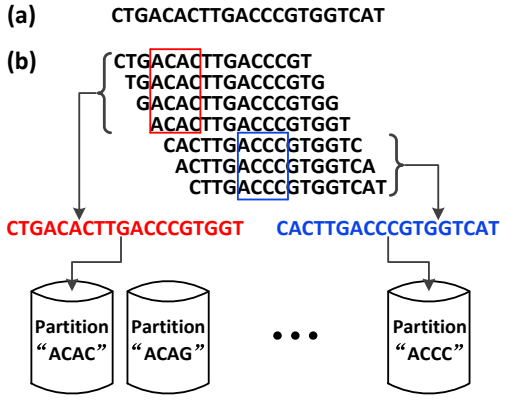

Fig. 3. A minimum substring partitioning example: (a)short read (b)K-mers and MSP process

The results of the Minimum Substring Partitioning is determined by the parameters k and p. Smaller p will increase the probability that consecutive k-mers share the same minimum p-substring and thus reduce the I/O cost. However, it will also introduce a problem where the distribution of partition sizes become skewed and the largest partition may not fit in the main memory. In the extreme case of $p = 1$, the size of the largest partition is almost as same as the size of the short reads data set and other partitions are almost empty (assuming the four nucleotides A, C, G, T are distributed randomly in the data set). In that case, we lose the point of partitioning. On the other hand, larger p will make the distribution of partition sizes

evener at the cost of decreasing the probability that consecutive k-mers share the same minimum p-substring and thus increasing the I/O cost. In the extreme case of $p \to k$, almost no adjacent k-mers will share the same minimum p-substring and thus no compression can be gained. Therefore one needs to make a tradeoff between the largest partition's size and the I/O overhead determined by p. Fortunately, as tested in our experiments, there is a quite wide range of values that p can choose without affecting the performance of MSP.

DEFINITION 7 (Wrapped Partitions). *Given a string set $\{s_i\}$, a hash function H, the user-specified number of partitions N, for any k-mer $s_{i,j}$, minimum substring partition wrapping assigns $s_{i,j}$ to the ($H(min_p(s_{i,j}))$ mod N)-th partition.*

Since each p-substring corresponds to one partition, the total number of partitions in MSP is equal to 4^p. When p increases, the number of partitions will increase exponentially and many partitions may become empty. To address this problem, one can introduce a hash function to wrap the number of partitions to any user-specified partition number. Then the k-mers are likely to be evenly distributed across partitions.

DEFINITION 8 (Minimum Substring with Reverse Complement). *Given a string s, a length-p substring t of s is called the minimum p-substring of s, if $\forall s'$, s' is a length-p substring of s or s' reverse complement, s.t., $t \leq s'$ ($\leq$ defined by lexicographical order).*

Definition 8 redefines minimum substring by considering the reverse complement. With this new definition, we can make sure each k-mer and its reverse complement k-mer are assigned to the same partition. This property can help us save much time and memory in the later processing (e.g. storing only the lexicographical smaller one of a k-mer and its reverse complement k-mer in hash table and avoiding reading each short read twice to explicitly process reverse complement) since a k-mer and its reverse complement are considered equivalent in bioinformatics and the information introduced by reverse complement can be inferred from the forward direction short reads. For simplicity reason, in the following discussions, if not mentioned explicitly, we will ignore the reverse complement issue. However, in our implementation and experiments, we do consider its impact.

4 METHODS

In this section, we describe the detailed methods to do k-mer counting and construct de Bruijn graph with the adoption of the minimum substring partitioning technique introduced in the last section.

4.1 MSP-based K-mer Counting

The first step is to partition short reads. In this step, we will cut each short read of length n into $(n - k + 1)$ k-mers and then dispatch these k-mers into different partitions. The Minimum Substring Partitioning technique introduced in Section 3 is used as our partitioning method. As mentioned before, with this partitioning method, we can compress consecutive k-mers dispatched to the same partition into one "super k-mer" to minimize the I/O cost.

A straightforward way to do minimum substring partitioning is as follows: (1) for each short read s, slide a window of width k through s to generate all $(|s| - k + 1)$ k-mers, (2) for each k-mer, find its minimum p-substring t, (3) generate "super k-mers" of s such that k-mers in each "super k-mer" share the same minimum p-substring. This method is very inefficient as it has to find the minimum p-substring for each k-mer, which introduces $(k - p + 1)$ p-substring comparisons. Let n be the length of s, since $k >> p$ and $n >> k$, this method needs to take $(k-p+1)*(n-k+1)$=$O(nk)$ p-substring comparisons.

The above solution does not utilize the inherent overlaps among k-mers and thus introduces many unnecessary comparisons. When we slide the window of width k through s, we can maintain a min-heap (Cormen *et al*., 1990) on p-substrings in the window. Each time, when we slide the window one nucleotide to the right, we delete the first p-substring in the previous window from the heap and insert the last p-substring of the current window into the heap. Whenever there is a change of the root node in the heap, a "super k-mer" will be generated. Since the number of p-substrings in the heap is $k - p + 1$ and the total number of p-substrings in s is $n - p + 1$, the number of p-substring comparisons in this approach is $O((n - p + 1)\log(k - p + 1))$=$O(n \log k)$.

The min-heap method is quite efficient in theory. However, the overhead introduced by the heap structure itself could be high in practice. As a tradeoff, here we use a simplified minimum substring partitioning algorithm whose worst case complexity is still $O(nk)$ but with a very good performance in real applications, as described in Algorithm 1.

Algorithm 1 Minimum Substring Partitioning

```
Input: String s = s_1 s_2 ... s_n, integer k, p.
min_s = the minimum p-substring of s[1, k]
min_pos = the start position of min_s in s
for all i from 2 to n − k + 1 do
  if i > min_pos then
    min_s = the minimum p-substring of s[i, i + k − 1]
    update min_pos accordingly
  else
    if the last p-substring of s[i, i + k − 1] < min_s then
      min_s = the last p-substring of s[i, i + k − 1]
      update min_pos accordingly
    end if
  end if
end for
```

As mentioned before, we can view k-mers generated in a way that a window with width k slides through a short read. In Algorithm 1, initially when the window starts at position 1, we scan the window to find the minimum p-substring, say min_s, and the start position of min_s, say min_pos. Then we slide the window forward, one symbol each time, till the right bound of the window reaches the end of the short read. After each sliding, we test whether the min_pos is still within the range of the window. If not, we have to re-scan the window to get new min_s and min_pos. Otherwise, we test whether the last p-substring of the current window is smaller than current min_s. If yes, we set this last p-substring as new min_s and update min_pos accordingly. If not, we just keep the old min_s to calculate the partition location. As described in last section, the neighboring k-mers will likely contain the same minimum p-substring. Therefore, the re-scan of the whole window will not occur very often. The worst case time complexity is $O(nk)$ p-substring comparisons.

Theorem 4.1 shows this algorithm is more efficient in practice (close to $O(n + lk)$) when s is broken to only a few number (l) of "super k-mers". This is very true in minimum substring partitioning of real short reads. Table 1 shows that the average number of breakdowns is small for several real short reads data sets.

Table 1. Average number of breakdowns for real short reads data sets

Data Set	n	k	p	Average Breakdown (l)
Budgerigar	150	59	10	5.22
Red tailed boa constrictor	121	59	10	3.89
Lake Malawi cichlid	101	59	10	2.77
Soybean	75	59	10	1.69

THEOREM 4.1. *Given a short read $s = s_1 s_2 \dots s_n$, $p, k \in N$, $p \leq k \leq n$, minimum substring partitioning breaks s into "super k-mers" $s[i_1, j_1]$, $s[i_2, j_2]$, ..., $s[i_l, j_l]$, $i_1 < i_2 < \dots < i_l$. Algorithm 1 needs at most $n + (l-1)k - pl + 1$ p-substring comparisons.*

PROOF. Algorithm 1 shows min_s and min_pos will change under two conditions: (1) $i >$ min_pos, or (2) the last p-substring of $s[i, i+k-1] <$ min_s. Under the first condition, we have to re-scan the k-mer $s[i, i+k-1]$ to get the new min_s, which introduces (k-p+1) p-substring comparisons. Under the second condition, we only have to compare the last p-substring of $s[i, i+k-1]$ with the current min_s, which only involves 1 p-substring comparison. Since the string s is broken into l "super k-mers", min_s and min_pos have changed for $(l-1)$ times. If all these $(l-1)$ changes are due to the first condition, the largest number of p-substring comparisons are needed. Therefore the total number of k-mer scans is l, including the initial scan of the first k-mer. These l k-mer scans will introduce $(k-p+1) * l$ p-substring comparisons. Within each of these l "super k-mers", $(n_m - 1)$ p-substring comparisons are required to test the second condition, where n_m is the number of k-mers within the "super k-mer" $s[i_m, j_m]$. For all l "super k-mers", the total number of p-substring comparisons due to this test is $\Sigma_{m=1}^{l}(n_m - 1) = (n - k + 1 - l)$. Therefore the total number of p-substring comparisons of Algorithm 1 is at most $(k-p+1) * l + (n-k+1-l) = n + (l-1)k - pl + 1$.

Note that in Algorithm 1, every time when we capture a minimum substring change at position j of s or we reach the end of s, we output a "super k-mer" of s that contains the previous minimum substring into the partition corresponding to that minimum substring. This part of code is not presented in Algorithm 1.

After obtaining the partitions, we can use a simple hash table whose keys are k-mers and values are k-mer counts to count the k-mer frequencies. For each partition, break the "super k-mers" into k-mers and insert these k-mers into a hash table. Since adjacent k-mers are only different by the first and last symbol, direct bit shift operations (A, C, G, T can be encoded using 2 bits) can be applied here to improve the performance. Whenever we see a new k-mer, we first look up the hash table to see if it is already in the hash table: if yes, we increase the frequency count by 1; otherwise, we put this k-mer along with an initial frequency value 1 into the hash table. After processing one partition, write the entries in hash table to a disk file and release the memory occupied by that hash table. Since all the occurrences of the same k-mer will locate in the same partition, the frequency count of a k-mer can be found in only one disk file. This is a very good property, as we do not have to later merge these frequency count disk files. The query of a k-mer's frequency is also very easy and efficient. Given a query k-mer, we can use MSP to calculate its partition location and then scan the corresponding count disk file to get the k-mer frequency.

4.2 MSP-based de Bruijn Graph Construction

DEFINITION 9 (Unipath). *Given a de Bruijn graph G, a path P in G is called unipath if it can be expressed as a concatenation of N consecutive adjacent k-mers $k_1, k_2, ..., k_N$ such that $k_1, k_2, ..., k_{N-1}$ have outdegree of 1 and $k_2, k_3, ..., k_N$ have indegree of 1, and P cannot be extended in either direction without violating these constraints. Two unipaths are called adjacent if the last k-mer of the first unipath is adjacent to the first k-mer of the second unipath.*

Assume we have partitioned a short read set using the Minimum Substring Partitioning method. In order to build a de Bruijn graph, we can construct a local de Bruijn graph first from each partition and then combine them to form a global graph. To reduce memory usage, we apply the concept of unipath, which is firstly introduced in AllPaths paper (Butler *et al*., 2008). A unipath can be regarded as a compression of k-mers with 1 indegree and 1 outdegree. After building unipaths in each partition, we only need to store the two end k-mers of each unipath and compress the other k-mers within the unipath into a string. By doing so, we are able to reduce the space usage without losing any information. The adjacency relationship among unipaths only relies on their end k-mers. Therefore in the global merging process, we load these end k-mers (and the string between them) in unipaths into memory and link them to get the global graph.

In order to build unipaths for each partition, we must have the complete neighborhood information of each k-mer in the partition. As described in k-mer counting, MSP may break a short read into several "super kmers" and dispatch them to different partitions. Therefore, for the first and last k-mers in each "super kmer," we may lose their neighborhood information. To address this issue, we extend the definition of Minimum Substring Partitioning.

DEFINITION 10 (Extended Minimum Substring Partitioning). *Given a string $s = s_1 s_2 \dots s_n$, $p, k \in N$, $p \leq k \leq n$, minimum substring partitioning breaks s to substrings with maximum length $\{s[i,j] | i+k-1 \leq j, 1 \leq i, j \leq n\}$, s.t., all k-mers in $s[i,j]$ share the same minimum p-substring t, and it is not true for $s[i, j+1]$ and $s[i-1, j]$. Then we will output the substring s[i-1, j+1] to the partition corresponding to the minimum p-substring t if $i - 1 \geq 1$ and $j+1 \leq n$. $s[i-1, j+1]$ is also called "extended super k-mer".*

With this extension, we can ensure that every k-mer in a partition will have complete information about its incoming edges and outgoing edges. Algorithm 2 shows the process of our de Bruijn graph construction method, which is performed after error correction. One important property of unipaths built from individual partitions is that they will be a portion of unipaths in the global de Bruijn graph. In comparison with a direct approach that builds the de Bruijn graph

Algorithm 2 MSP-based de Bruijn graph construction

Input: Short reads set $\{s\}$
do extended minimum substring partitioning on $\{s\}$
for all minimum substring partitions **do**
  build unipaths
  write unipaths (end k-mers along with compressed strings) into a disk file
  release the memory
end for
load end k-mers together with compressed strings in all the partitions into memory and merge them based on their overlaps

from scratch, Algorithm 2 significantly reduces memory consumption since it compresses k-mers on the unipaths in each partition first, before loading all the partitions into memory. In contrast, the direct approach cannot do the compression on the fly before loading all the k-mers into memory.

5 EXPERIMENTAL RESULTS

In this section, we present experimental results that illustrate the efficiency of our MSP-based k-mer counting and de Bruijn graph construction method on four large real-life short reads data sets: Budgerigar (bird), Red tailed boa constrictor (snake), Lake Malawi cichlid (fish) and soybean. (1) We analyze the efficiency of our k-mer counting method by reporting the memory and time costs and compare it with two state-of-the-art k-mer countering algorithms: Jellyfish (Marcais *et al*., 2011) and BFCounter (Melsted *et al*., 2011); (2) We then investigate the efficiency of our de Bruijn graph construction method by discussing the memory usage and run time and compare it with two de novo assemblers: Velvet (Zerbino *et al*., 2008) and SOAPdenovo (Li *et al*., 2010a). All the experiments, if not specifically mentioned, are conducted on a server with 2.40GHz Intel Xeon CPU and 24 GB RAM, using a single thread.

5.1 Sequence Datasets

Four very large real-life short read sequence data sets are used to test our algorithms. The first one is the sequence data of Budgerigar (bird) downloaded from `bioshare.bioinformatics.ucdavis.edu/Data/hcbxz0i7kg/Parrot/BGI_illumina_data/`. These short reads were sequenced from the Illumina HiSeq 2000 technology. The second one is the sequence data of Red tailed boa constrictor (snake) downloaded from `bioshare.bioinformatics.ucdavis.edu/Data/hcbxz0i7kg/Snake/short_inserts/`. These short reads were obtained with the Genome Analyzer technology. The third one is the sequence data of Lake Malawi cichlid (fish) downloaded from `bioshare.bioinformatics.ucdavis.edu/Data/hcbxz0i7kg/fish/`. And the last one is the sequence data of soybean downloaded from `ftp://public.genomics.org.cn/BGI/soybean_resequencing/fastq/`. Some basic facts about these four data sets are shown in Table 2.

Table 2. Basic facts about the four sequence data sets used in our experiments

	bird	snake	fish	soybean
Format	fastq	fastq	fastq	fastq
Size (GB)	106.8	181.7	137.4	40.1
Avg Read Length	150	121	101	75
No. of Reads (million)	323	573	598	227

5.2 K-mer Counting Efficiency

We conduct experiments to compare our MSP-based k-mer counting method with two state-of-the-art k-mer counting algorithms: Jellyfish, which is a fast, memory efficient k-mer counting tool based on a multi-threaded, lock-free hash table optimized for counting k-mers up to 31 nucleotides in length; BFCounter, which is a k-mer counting tool with greatly reduced memory requirements based on bloom filter, a probabilistic data structure. For all the three methods, we set the number of threads to 1, since BFCounter only supports single thread. For our method, we set the number of wrapped partitions to 1,000 (to reduce memory footprint) and the minimum substring length p to 10.

The memory usage and run time for counting 31-mers on short reads of the Budgerigar data set for various levels of coverage are shown in Figures 4 and 5, respectively. Note that in order to get different levels of coverage, we randomly sampled the short reads data set to obtain a desired amount of sequence.

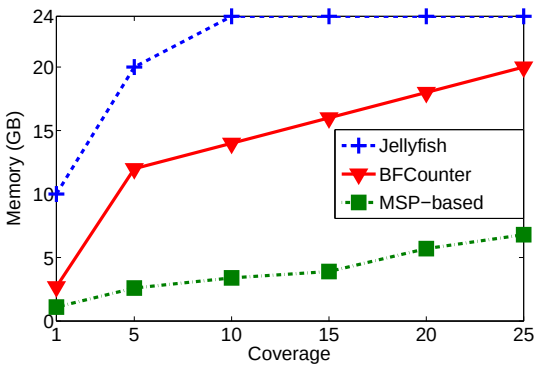

Fig. 4. Memory usage of the three counting algorithms for various levels of coverage on the Budgerigar data set when counting 31-mers

Figures 4 and 5 show that when the coverage is low (e.g. less than 5), Jellyfish is slightly quicker than our method. The reason is that in a low coverage situation, Jellyfish is able to finish all the computation in memory. Therefore it becomes a purely memory-based algorithm in this case. In contrary, our method relies on the disk-based minimum substring partitioning, so it is not going to outperform Jellyfish in this case. However, even in a low coverage situation, our method is still much faster than BFCounter and uses

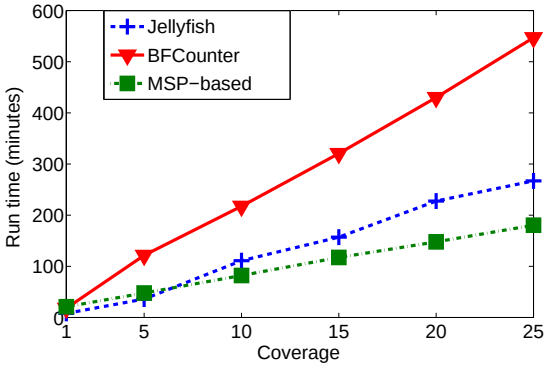

Fig. 5. Run time of the three counting algorithms for various levels of coverage on the Budgerigar data set when counting 31-mers

much less memory than both Jellyfish and BFCounter. As the coverage increases, our method quickly dominates the scene. In a high coverage situation, the main memory is not big enough for Jellyfish to finish all the computation in memory and therefore it has to write intermediary results to disk and later merge them. This gives our method a chance to outperform Jellyfish in both memory and time. BFCounter has the advantage that its memory usage does not increase a lot as the coverage increases. However, compared with our method, it still requires more memory and much longer running time to finish the task.

Table 3 presents the memory usage and run time for the three algorithms when applied to the other three data sets (the snake, fish and soybean data sets) for counting 31-mers.

Table 3. Comparison of memory usage and run time for counting 31-mers on the snake, fish and soybean data sets.

Algorithm	Memory (GB)			Run Time (hours)		
	snake	fish	soybean	snake	fish	soybean
Jellyfish	24	24	24	11.2	6.6	1.4
BFCounter	>24 [1]	>24 [2]	13	>20 [1]	>20 [2]	3.7
MSP-based	9.6	10.5	6.5	6.1	4.5	1.2

As can be seen from Table 3, when applied to a large sequence data set with deep coverage, our method soon demonstrates its advantages. It uses less memory than both Jellyfish and BFCounter. Actually BFCounter ran out of memory and was unable to finish successfully on the snake and fish data sets. Jellyfish was able to finish the task by writing intermediary results to disk and later merging them. But unfortunately, its merging process is relatively expensive and therefore it is slower than our method, which requires no additional merging steps after partial results are generated from individual partitions.

[1] On a 96 GB memory machine, BFCounter can finish counting 31-mers on the snake data set using 24.2 hours and 39.7 GB memory.

[2] On a 96 GB memory machine, BFCounter can finish counting 31-mers on the fish data set using 21.3 hours and 32.2 GB memory.

5.3 de Bruijn Graph Construction Efficiency

Next we conduct experiments to illustrate the efficiency of our MSP-based de Bruijn graph construction method. In this set of experiments, we use a server with 2.40GHz Intel Xeon CPU and 96 GB RAM since the last step of de Bruijn graph construction needs to put all the compressed unipaths into memory and link them, which may require more than 24GB RAM. We compare our method with two state-of-the-art assembling algorithms: Velvet, a classic de Bruijn graph based assembler and SOAPdenovo, a highly optimized assembler. We ran experiments using only one thread for all the three algorithms with k-mer length being 59. Again, to reduce memory footprint, we set the number of wrapped partitions to 1,000 and the minimum substring length p to 10 for our MSP-based de Bruijn graph construction method.

Table 4 presents the memory usage and run time for three de Bruijn graph construction algorithms when applied to the four data sets with k being 59.

Table 4. Comparison of memory usage and run time for building de Bruijn graph on the bird, snake, fish and soybean data sets

Algorithm	Memory (GB)				Run Time (hours)			
	bird	snake	fish	bean	bird	snake	fish	bean
Velvet	>96	>96	>96	77	N/A	N/A	N/A	2.7
SOAPdenovo	>96	>96	>96	60	N/A	N/A	N/A	2.9
MSP-based	45	43	65	12	5.1	5.8	8.8	1.2

Table 4 shows that our method can finish building de Bruijn graph efficiently with the use of a reasonable amount of memory for very large sequence data sets (the most memory intensive part is the last "link" step). As a comparison, Velvet and SOAPdenovo quickly used up all the available memory and thus failed to build the de Bruijn graph successfully on the bird, snake and fish data sets. On the relatively small soybean data set, Velvet and SOAPdenovo were able to finish successfully; but both of them consumed much more memory than our method.

6 DISCUSSION

6.1 Parallelizability

Our MSP-based k-mer counting method and de Bruijn graph construction method can be parallelized to support multi-threads or be distributed to multiple machines to do parallel processing. There are three distinct phases in the Minimum Substring Partitioning process. First, it reads the short read sequences. Second, it calculates the minimum substring of each k-mer and merges possible adjacent k-mers into "super k-mers". Last, it writes the "super k-mers" back to disk files. Phase 1 and phase 3 are I/O operations, so the speedup

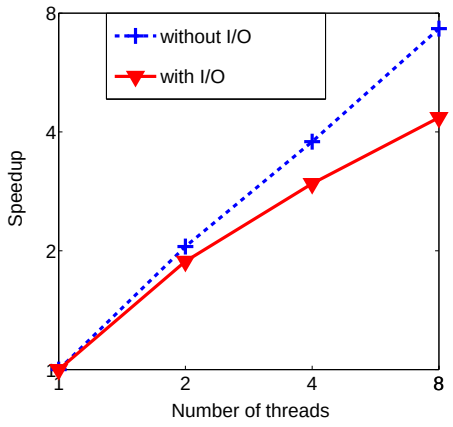

Fig. 6. Speedup versus number of threads for MSP-based k-mer counting

can be obtained by using multi-threads to process phase 2. After partitioning, different partitions are completely disjoint. Therefore it is helpful to use different threads to process different partitions simultaneously. We implemented a preliminary multi-thread version of our k-mer counting method. Figure 6 shows the speedup obtained with the increasing number of threads. Here k-mers are counted on 5 times coverage of the Budgerigar short reads data set with $k = 31$ and the figure uses logarithm on both axes. From Figure 6 we can see that: without considering the I/O operations, our method has an almost linear speedup up to 8 threads (our test machine has 2 cpu, each with 4 cores), indicating a good parallelizability; when including the I/O operations, the speedup is almost linear up to 2 threads and then levels off since the CPU calculation is already fast enough and the I/O bandwidth becomes the main bottleneck of our counting method.

6.2 Future Work

There are some future avenues to pursue to further improve our work. First, we can adopt the techniques (e.g. variable length encoding) introduced in Jellyfish (Marcais *et al.*, 2011) to make space-efficient encoding of keys and reduce the memory usage of each hash entry to further reduce the memory consumption. Second, we can make use of the entropy compression technique (or succinct data structure) introduced in (Conway *et al.*, 2011) to get a nearly optimal compressed representation of the de Bruijn graph, which may help decrease the memory cost of the "link" step in our graph construction method. Third, we can think about extending the use of MSP from building de Bruijn graph to the whole assembly process. Since the k-mers in different MSP partitions are completely disjoint and the majority of adjacent k-mers in original short reads are retained in the same partition, it is possible to perform local assembly (including some error correction steps like tip removal and bubble merging) for each partition and later "glue" these local assembly results to obtain the global assembly results. By doing so, the whole assembly process can be done with very small memory footprint and speed up a lot with the help of parallel assembly of multiple partitions.

7 CONCLUSION

In this paper, we aimed at the computational bottlenecks in k-mer counting and de Bruijn graph construction, which are two very important tasks in genome sequence assembly. We proposed a disk-based approach based on the novel concept, Minimum Substring Partitioning (MSP), to solve the memory overwhelming problem. MSP breaks the short reads into multiple disjoint partitions so that each partition only requires a very small amount of memory to process. By leveraging the overlaps among the k-mers derived from the same read, MSP is able to achieve astonishing compression ratio so that the I/O cost can be greatly reduced, making the method be very efficient in terms of time and space. Our MSP-based k-mer counting algorithm and de Bruijn graph construction algorithm were evaluated on real DNA short read sequences. Experimental results show that they can not only successfully finish the tasks on very large data sets within a reasonable amount of memory, but also achieve better performance than the existing algorithms.

ACKNOWLEDGEMENT

We thank Dr. Michael Schatz in the Cold Spring Harbor Lab for the initial discussion on assembling large genomes. The Assemblathon website (`http://assemblathon.org`) provides us the Budgerigar (bird), the Red tailed boa constrictor (snake) and the Lake Malawi cichlid (fish) data, and Beijing Genomics Institute provides the soybean data. The code of Velvet and SOAPdenovo are generously provided by Dr. Daniel Zerbino and Beijing Genomics Institute, respectively. We also thank Dr. Robin Zhou at UCSB, Dr. Yu S. Huang at UCLA, and Russell McLoughlin at Lawrence Livermore National Lab for experiments and applications of de novo sequence assembly.

Funding: This research was sponsored in part by the U.S. National Science Foundation under grant IIS-0847925 and IIS-0954125. The views and conclusions contained herein are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notice herein.

REFERENCES

Mardis, E.R. (2008) Next-generation DNA sequencing methods, *Annu. Rev. Genomics Hum. Genet.*, **9**, 387-402.

Myers, E.W. *et al.* (2000) A whole-genome assembly of Drosophila, *Science*, **287**, 2196-2204.

Batzoglou, S. *et al.* (2002) ARACHNE: a whole-genome shotgun assembler, *Genome research*, **12**, 177-189.

Havlak, P. *et al.* (2004) The Atlas genome assembly system, *Genome research*, **14**, 721-732.

Mullikin, J.C. and Ning, Z. (2003) The phusion assembler, *Genome research*, **13**, 81-90.

Platt, D. and Evers, DJ (2010) Forge: A Parallel Genome Assembler Combining Sanger and Next Generation Sequence Data, *http://combiol.org/forge/*.

Pevzner, P.A. *et al.* (2001) An Eulerian path approach to DNA fragment assembly, *Proceedings of the National Academy of Sciences*, 9748-9753.

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

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

Butler, J. *et al.* (2008) ALLPATHS: de novo assembly of whole-genome shotgun microreads, *Genome research*, **18**, 810-820.

Li, R. *et al*. (2010) De novo assembly of human genomes with massively parallel short read sequencing, *Genome research*, **20**, 265-272.

Miller, J.R. *et al*. (2010) Assembly algorithms for next-generation sequencing data, *Genomics*, **95**, 315-327.

Li, R. *et al*. (2010) The sequence and de novo assembly of the giant panda genome, *Nature*, **463(7279)**, 311-317.

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

Melsted, P. and Pritchard, J.K. (2011) Efficient counting of k-mers in DNA sequences using a bloom filter, *BMC Bioinformatics*, **12**, 333-339.

Cormen, T. *et al*. (1990) *Introduction to Algorithms*. Chapter 6. MIT Press. Cambridge, MA.

Conway, T.C. and Bromage, A.J. (2011) Succinct data structures for assembling large genomes, *Bioinformatics*, **27(4)**, 479-486.

Salzberg, S.L. *et al*. (2012) GAGE: A critical evaluation of genome assemblies and assembly algorithms, *Genome research*.

Earl, D. *et al*. (2011) Assemblathon 1: A competitive assessment of de novo short read assembly methods, *Genome research*, **21**, 2224-2241.

Iqbal, Z. *et al*. (2012) De novo assembly and genotyping of variants using colored de Bruijn graphs, *Nature Genetics*.