Parallel Implementation of De Bruijn Graph

Alex Cao[1]

[1]Simon Fraser University, CMPT 431.

Keywords: Parallel Programing, Distributed System, Bioinformatics, C++

1 Introduction

As an intersecting field of biology, computer science, and mathematics, bioinformatics continuously offers advanced tools and approaches in understanding the complexity of biological data. This field leverages computational techniques to solve biological problems as modern experimental techniques like high-throughput sequencing are producing vast amounts of data. Among all the available computational tools utilized in bioinformatics, the De Bruijn graph represents a pivotal innovation in genomics sequencing and assembly and had continuously been one of the more popular sequencing tool in the field.

The next generation sequencing (NGS) technologies has revolutionized biology by allowing rapid and cost-effective generation of large-scale genomics data [1]. However, because NGS is a short read sequencing by breaking the genome into small fragments, around 50 to 300 bases in size [1], this will produces a large volume and fragmented data posed a challenges for assembly those short, overlapping reads. Because of this, De Bruijn Graph had became one of the critical tool to enable efficient representation and processing of sequencing data in reconstructing genomes with high accuracy and efficiency [2].

De Bruijn graph is a directed graph that had been designed to well adapt the representing overlaps between sequences. The graph is made up with vertices and nodes where the vertices in the graph represent sequences of length k (known as k-mers), and edges denote the overlap of $k-1$ nucleotide between consecutive k-mers [3]. This precise overlapping allows the De Bruijn graph to understand the connectivity and continuity in a genomics data. By mapping out the relationships between each of the short reads, it becomes a tractable problem of graph traversal where Eulerian paths or cycles within the graph can be identified [3].

An Eulerian path is a pathway that visited every edge exactly once, which represents a sequence that incorporates every k-mer overlap. To allow for this path to be constructed, De Bruijn graph include a specific algorithms tailored to this purpose [4]. One of most common algorithms to achieve this task is Hierholzer's algorithm [5]. Hierholzer's algorithm utilizes a similar approach to Depth-First Search (DFS), an algorithm that traversing tree or graph data by starting at a root node and explores as far as possible along each branch before backtracking [5].

In Hierhlozer's algorithm, it first initializes from a random node and proceeds to explore edges in a DFS manner. It will continue exploring edges until it forms a closed loop or cycle [5]. If the graph is to determined as non-Eulerian or the cycle does not pass all edges, Hierholzer's algorithm will start another round of search at a new starting point. This still is repeated, merging cycles to form a larget cycle, until all edges in the graph are visited [5].

Because of how Hierholzer's algorithm is designed, it comes with various challenges when trying to parallelize it to enhance performance. The core difficulty lies in the inherently sequential nature of the

Eulerian path problem. In a De Bruijn graph, each vertex represetns a k-mer (a string of k nucleotides), and each directed edge represetns an overlap of k-1 nucleotides between two kmers. Imagine a simple De Bruijn graph with 4 nodes, A, B, C, D and edges representing overlaps. To find an eulerian path, one must start from a specific node and follow the edges in a way that every edge is visited exactly once. This process is inherently sequential because when visiting node B from A, it dictates that the next visited should be C or D given edges.

This also means that one cannot just simply divide the graph in subgraphs without losing the context of global connectivity. Since each edge's visitation is dependent on the visitation of its preceding edges in the path, it requires extensive coordination to partition the graph. Also it is a non-trivial problem to connect path within a subgraph as each might find a valid Eulerian subpath and reorder and connect these paths correctly without violating the Eulerian constrains is difficult.

2 Background

The current state of the art in serial implemenation of de Bruijn graph includes a variety of methods all with a focus on enhancing the efficiency, scalability, and accuracy of genomic data analysis. One of the more notable improvement made to the serial implementation is in the tool Cuttlefish 2 [6]. This tool is designed for constructing compacted de Bruijn graphs from a collection of reference sequences or raw sequencing reads. In Cuttlefish 2 [6], the focus was on optimizing data processing through a streamlined approach that discards potential sequencing errors by working on the k+1 mer spectrum resulting in reduces time and memory required for constructing a maximal unitigs of the graph [6].

Another notable advancement is the introduction of simplitigs showcased in the tool ProphAsm [7]. The idea behind simplitigs is that it offers a compact representation by covering the graph with a set of maximal non-overlapping paths, therefore optimizing both the number of sequences and their cumulative length [7]. Simplitigs are created through the use of a greedy algorithm that extends sequences in both direction minimizing memory footprint [7].

When it comes to Parallel and Distributed implementations of de Bruijn graphs also seek innovations that aimed at optimizing memory usage while ensuring accuracy. One approach is utilizing a Bloom filter in combination with a critical false positive structure (cFP) to obtain a space-efficient and accurate representation of the de Bruijn graph [8]. The core idea of this method is to insert all k-mers into a Bloom filter and deduces edges by querying for all possible extensions of a k-mer [8]. The cFP assist in avoiding false branching caused by the inherent false positives of the Bloom filter [8].

Another tool is ABySS, produced by researcher at BC Genome Science Centre. ABySS focus on optimizing the assembly of large genomes through parallel and distributed implementation [9]. This tool is notable for its use of MPI which allows it to run efficiently on both shared-memory and distributed-memory computer clusters [9]. By using k-mer pairs separated by a fixed-size-gap, ABySS can span larger sequences without additional memory overhead, offering improved scalability for long reads [9]. Additionally, tools like Konnector and Sealer are integrated into ABySS to enhance its functionality [9].

Overall, for de Bruijn graph, being a popular method for genome sequencing, advancements are made in both serial and parallel/distributed implementation, addressing the growing demands for efficient and scalable data processing tools. These developments all improve the accuracy and efficiency of genome assembly in their own way, allowing the de Bruijn graph to be applicable in many different scenarios.

3 Method

3.1 Serial Implementation

For my serial version of De Bruijn Graph algorithm, I decided to implement Hierholzer's algorithm, inspired by https://colab.research.google.com/github/BenLangmead/comp-genomics-class/blob/master/notebooks/CG_deBruijn.ipynb. This is an efficient algorithm and had been a popular approach to transverse through the graph and finding eulerian path or cycles. Because the Hierholzer's algorithm

can methodically tranverse throught every edge exactly once, the linear time is relative to the number of edges in the graph.

Algorithm 1 Serial: Construct De Bruijn Graph

```
Ensure k ¿ 1, strIter is not empty
Initialize nsemi, nbal, nneither to 0, head and tail to nullptr
For each st in strIter:
    If circularize, append the first k-1 characters of st to st
    For each i from 0 to st.size()-(k-1):
        Extract kmer, km1L, km1R from st
        If km1L or km1R in nodes, set nodeL or nodeR to nodes[km1L] or nodes[km1R], else create new Node and add to nodes
        Increment nodeL-¿nout and nodeR-¿nin
        Add nodeR to G[nodeL]
For each kv in nodes:
    If node is balanced, increment nbal
    Else if node is semi-balanced, set tail to node if node-¿nin == node-¿nout+1, set head to node if node-¿nin == node-¿nout-1
    Else, increment nneither
```

This function is a serial implementation for constructing a De Bruijn graph from a list of strings. The algorithm will create nodes for each unique (k-1)-mer in the strings and edges for each k-mer to represent the overlaps between the (k-1)-mers in the strings.
The `circularize` parameter determines whether the function should treat each string as a circular sequence. If `circularize` is true, the function appends the first (k-1) characters of each string to the end of the string, wrapping around the sequence. But because this is a simple implementation of De Bruijn Graph, this will never be use.
The function also keeps track of the "balance" of each node. If a node is balanced, it means that the in-degree and out-degree of that node equals. A node can also be "semi-balanced" which means that if the degrees differ by one, and "neither" otherwise. The function keeps counters for the number of each type of node and also keeps track of the "head" and "tail" nodes, which are the semi-balanced nodes.

Algorithm 2 Serial Eulerian Walk or Cycle

```
Ensure Graph G is initialized
Initialize an empty unordered map g
For each kv in G, add kv.second to g[kv.first]
Set src to the first key in g
Initialize an empty vector tour
If isEulerian():
    If hasEulerianWalk(), add edge from tail to head in g
    Else, set src to the first key in g
Call euler_r with g, src, and tour
Remove the last element from tour
If isEulerian() and hasEulerianWalk(), rotate tour so that head is at the beginning
Return tour
```

In this algorithm, Hierholzer's algorithm is used to construct an eulerian path or cycle from a De Bruijn graph. It first translate the graph into a more accessible format, mapping each vertex to its adjacent edges. Hierholzer's algorithm is then applied, where it will first determine a starting point, follow by recursively visits every edge exactly once, building the path or cycle. If the graph is an eulerian walk, it will adjusts the path to start and end at specific vertices.

Algorithm 3 Recursive Eulerian Walk or Cycle Helper Function

```
Ensure Kmer and tour are initialized, node is a valid node
While kmer[node] is not empty:
    Set next_node to the last element of kmer[node]
    Remove the last element from kmer[node]
    Call euler_r with kmer, next_node, and tour
Append node to tour
```

3.2 Parallel Implementation C++ Threads

The parallel implementation of De Bruijn graph using C++ threads is an attempt to increase the performance. Although the focus of paralleling De Bruijn graph is focused on the traversal of the graph, my implementation included paralleling the constructing graph part of the algorithm as well to future increase speed.

Algorithm 4 C++ Threads: Construct De Bruijn Graph

```
Ensure k ¿ 1, seqs is not empty, nThreads ¿ 0
Initialize nsemi, nbal, nneither to 0, head and tail to nullptr
Create a vector of threads of size nThreads, a mutex, and a vector of timers of size nThreads
For each t from 0 to nThreads:
    For each String in Seqs:
        If circularize, append the first k-1 characters to kmers
        For each i from 0 to kmers.size()-(k-1):
            Extract kmer, km1L, km1R from st
            Initialize nodeL and nodeR to nullptr
            Lock Mutex
            If km1L or km1R in nodes, set nodeL or nodeR to nodes[km1L] or nodes[km1R], else create new Node and add to nodes
            Unlock Mutex
            Increment nodeL-¿nout and nodeR-¿nin
            Lock Mutex, add nodeR to G[nodeL], Unlock Mutex
Join all threads
For each kv in nodes, follow the same steps as in the serial implementation
```

In this function, the parallelization for constructing De Bruijn graph is achieved by first partition sections of sequences to different threads and having each of the thread will process nodes and edges in parallel. A mutex, mtx, is used to protect shared data from beng simultaneously accessed by multiple threads to prevent data races and inconsistent result. In this function, the shared variables are nodes map and the G graph, so mutex was locked whenever a thread tries to change the two variables. After All chunk are finished processed, the function waits for all threads to finish using join method, ensuring that the graph construction is complete. The result of the function is the same as the serial implementation.

Algorithm 5 C++ Threads: Eulerian path or cycle

```
Initialize empty map g with nodes and edges from G
Set src to first key in g
If isEulerian() and hasEulerianWalk(), add edge from tail to head in g
For each i in nThreads, set headNodes[i] to i-th key in g
For each thread i:
    Set node to headNodes[i]
    Call euler_r_parallel with node and paths[i]
Join all workers
Initialize tour, included
For each node in paths[i], add node to tour and included
Return tour
```

Algorithm 6 C++ Threads: euler_r_parallel recursion helper function

```
While true:
        If g[node] is empty, erase node from g and break
        If stop[i] is true and g is empty, break
        Otherwise, set next_node to last node in g[node] and remove it
         For each j in headNodes, if next_node equals headNodes[j] and j != i, set stop[i] to true and
    break
Lock pathMutexes[i], add node to tour, unlock pathMutexes[i]
```

As mentioned in Intruduction, when it comes to parallelizing the Eulerian path search using Hierholzer's algorithm, some challenges occurs. To over come those challenges, this C++ parallel version of the algorithm enhances Hierholzer's algorithm for finding Eulerian path through a Dynamic Task Allocation strategy. The algorithm first divide the graph into segments for each thread to handle independently. Each thread starts at a random node denoted as head nodes and initialize Hierholzer's algorithm separately. Once a thread reaches the head node of another thread, it will dynamically adjust their paths by joining the path of the first thread to the second thread, and the first thread will seek a new node to start a new round of Hierholzer's algorithm.
To ensure shared datas are consistent without race considtions, mutex are used to lock visited (list of visited node), and tour (list of path) whenever a thread tries to modify it.

3.3 Distributed Implementation

When it comes to distributed implementation of De Bruijn Graph using Message Passing Interface (MPI) in C++, only the traversing the graph portion is parallelized. I did not parallelize the construction part of De Bruijn graph involves several consideration. Firstly, to construct the graph, my code used custom data structures like Nodes and Graph and MPI is not designed for destribute shared memory systems like it. In order to implement MPI for constructing the graph, each of the world rank would need to construct its own graph by partitioning the input sequences, and the join the graph by the end. This creates a problem where MPI can only send and recieve integer, floating point, complex number, byte, character, and logical type, so in order for it to send a class, it need to serialize then deserialize and this process is pron to error. Even if serializing and deserializing is able to maintain the subgraph structure, when joining all the subgraph each world rank still need to iterate through all nodes to ensure there are no errors like duplication, and missed edges. And because of my parallel implementation of Hierholzer's algorithm, each world rank need to have a copy of the graph, therefore in term of computation and resources, it is not idead to implement MPI for graph building.

Algorithm 7 MPI: Eulerian Path or Cycle

```
Initialize empty map g
For each node src and its adjacent nodes dst in G, add dst→km1mer to src→km1er vector in G
If g is empty, return []
Initialize src to km1mer of first node
If isEulerian & isEulerianWalk, add edge from tail→km1mer to head→km1mer
Initialize visited[], paths, stacks, stop(0), headNodes with size of rank
Repeat until no unvisited nodes in g:
    Find an unvisited node, mark it as visited
    MPI Barrier
    Broadcast the node
    Receive node, assign to headNodes[rank]
    Call euler_r_parallel(...)
MPI Barrier
If rank==0, receive path from other ranks
```

Algorithm 8 MPI: eulerian function

```
Initialize map g with nodes and edges from G
If g is empty, return []
If g is empty, return Initialize src, visited[], paths, stacks, stop(0), headNodes
If isEulerian & isEulerianWalk, add edge
Repeat until no unvisited nodes in g:
    Push node to stack
    While stack is not empty and stop[rank] is false:
        Pop node from stack
        If node is in g, insert node to tour and push adjacent nodes to stack
        Broadcast each adjacent node to all processes
        Remove node from g
        For each headNode in headNodes:
            If node equals headNode and rank is not equal to current rank, broadcast signal, rank, and current rank to all processes
            Find a new unvisited node, mark it as visited, and broadcast it and the visited node to all processes
            Insert paths[rank] to paths[j], clear paths[rank], and push new node to paths[rank]
            If received signal is 1, set stop[received_rank] to 1, insert paths[received_rank] to paths[received_j], and clear paths[received_rank]
        If stop[rank] is true, break the loop
        Broadcast stop[rank] to all processes
If stop[rank] is true, abort the MPI execution
```

This function parallelizes Hierholzer's algorithm for finding Eulerian walk using MPI in C++. The main concept is similar to previous implementation. With MPI, MPI barrier is used to ensure all world ranks are synchronized up until that point. This is necessary because any changes made to "global" variable need to be sychronized. The function then loop around to find non-visited nodes broadcast to all processes and start traversing from that node. The dfs parallel function performs a DFS where if it visited a head node of another process, it merge path with that process and finds a new node that has not been visited to start traversing again. MPI broadcast is used to broadcase the current node and the head nodes of each process's path. Lastly, MPI Iprobe and recv is used to collect all path found by each process. MPI Iprobe checks if a message is avaliable, because in therory there should be only one process have all the node and MPI recv receives the message.

4 Result

The following are the results shown in term of time, the number of node each process process, and the accuracy when use BLAST. BLAST stands for Basic Local Alignment Search Tool, and it is used for

Test Case 1	Serial Version	Thread Version	MPI Version
De Bruijn Graph Building Time (s)	5.65308	6.30648	6.154328
		0.274964	6.154331
		0.133514	6.154323
		0.0215938	6.154326
Total Graph Building Time (s)		6.6079	6.154328
Nodes/Edges	2104181/2103948	2104181/2103948	2104181/2103948
Eulerian Path Traversing Time (s)	3.89893	0.0515251	14.792495
		2.38E-05	14.77984
		7.61E-05	14.791732
		8.82E-06	14.795266
Total Eulerian Path Time (s)		5.75221	14.795266
Eulerian Path Length Size	18809	17411	9758
Average Time Per Length (s/len)	0.00020729065	0.00033037792	0.0015162191
Total Time (s)	11.0554	12.3623	20.949818

Table 1 Table 1: We can see that for test case 1, all three implementations produced a graph with nodes of 2104181 and edges of 2103948. Although the graph constructed seems to be consistent but the threads version is around 15% slower while the Distributed version is around 10% slower. When it comes to traversing through the graph, the overall speed of threads and distributed version are slower than serial version, which can be reflected on the total time taken for the program.

Test Case 1 BLAST Result	Scientific Name	Max Score	Total Score	Query Cover	E value	Per. Ident	Acc. Len
Serial	Escherichia coli	34659	34659	99%	0.0	99.93%	4764972
Threads	Escherichia coli	32071	33021	100%	0.0	99.73%	5302688
MPI	Escherichia coli	17921	18976	100%	0.0	99.82%	4663521

Table 2 Table 2: We can see that all three versions of De Bruijn Graph algorithm returns comparative scores when it comes to Query Cover, E value, Percentage Identity and Accession Length, but Serial version has the highest score overall, with the max score and total score while Threads version is second and MPI version is last.

Test Case 2	Serial Version	Thread Version	MPI Version
De Bruijn Graph Building Time (s)	4.74005	8.07779	5.171531
		6.05135	5.171529
		1.60965	5.171527
		0.260198	5.171530
Total Graph Building Time (s)		8.31924	5.171530
Nodes/Edges	1687080/1686975	1687080/1686975	1687080/1686975
Eulerian Path Traversing Time (s)	3.15469	0.00618315	14.726784
		7.79629e-05	14.779012
		0.000102997	14.732384
		9.29832e-05	14.697435
Total Eulerian Path Time (s)		4.62833	14.779012
Eulerian Path Length Size	1767	2267	11845
Average Time Per Length (s/len)	0.00178533672	0.00204161005	0.00124770046
Total Time (s)	9.06993	12.9493	19.951001

Table 3 Table 3: We can see that for test case 1, all three implementations produced a graph with nodes of 1687080 and edges of 1686975. Although the graph constructed seems to be consistent but the threads version is around 200% slower while the Distributed version is around 20% slower. When it comes to traversing through the graph, the overall speed of threads are slower than serial version, but when look at the average time for each program to process Eulerian path per length, distributed implementation is faster than serial version, but the total time, it is the slowest.

analyzing and sequences of genes and proteins from research to known database. Two of the bigger test cases are obtained from NSBI SRC, while the smaller test cases are from previous class. Please note that because this is a simple implementation of De Bruijn Graph, it does not include any form of error correction which is often the case in real life examples. Therefore no matter if the sequence is Eulerian path or not, it will still output a path for comparison.

Test Case 2 BLAST Result	Scientific Name	Max Score	Total Score	Query Cover	E value	Per. Ident	Acc. Len
Serial	Staphylococcus aureus	3445	3445	100%	0.0	100.00%	2784836
Threads	Staphylococcus aureus	4338	9994	100%	0.0	99.79%	2879954
MPI	Staphylococcus aureus	21874	23664	100%	0.0	100.00%	2879954

Table 4 Table 4: We can see that all three versions of De Bruijn Graph algorithm returns comparative scores when it comes to Query Cover, E value, Percentage Identity and Accession Length, but Distributed version has the highest score overall, with the max score and total score while Threads version is second and serial version is last.

Test Case 3	Serial Version	Thread Version	MPI Version
De Bruijn Graph Building Time (s)	0.00671482	0.0164359	0.007333
		0.018188	0.007365
		0.0179489	0.007328
		0.018044	0.007383
Total Graph Building Time (s)		0.0187778	0.007383
Nodes/Edges	4656/4655	4656/4655	4656/4655
Eulerian Path Traversing Time (s)	0.008744	0.00756383	0.834106
		3.91006e-05	0.832049
		0.0095799	0.832774
		0.00177312	0.832533
Total Eulerian Path Time (s)		0.019429	0.834106
Eulerian Path Length Size	4661	4662	4661
Average Time Per Length (s/len)	0.00000187599	0.00000416752	0.0001789543
Total Time (s)	0.016434	0.038307	0.842200

Table 5 Table 5: We can see that for test case 1, all serial and MPI version implementations produced a graph with nodes of 4656 and edges of 4655. Although the graph constructed seems to be consistent but the threads version is around 300% slower while the Distributed version is around 10% slower.
For the Eulerian Path Length, Thread version is inconsistent with others by having 1 extra length. When it comes to the speed traversing through the graph, the overall speed of threads and distributed version are slower than serial version, which can be reflected on the total time taken for the program.

Test Case 3 BLAST Result	Scientific Name	Max Score	Total Score	Query Cover	E value	Per. Ident	Acc. Len
Serial	Homo sapiens DNA	3544	9.004e+05	98%	0.0	89.32%	530332
Threads	Homo sapiens DNA	2634	4.364e+05	98%	0.0	99.65%	530332
MPI	Homo sapiens DNA	2615	6.182e+05	98%	0.0	99.38%	530332

Table 6 Table 6: We can see that all three versions of De Bruijn Graph algorithm returns comparative scores when it comes to Query Cover, E value, Percentage Identity and Accession Length, but Serial version has the highest score overall, with the max score and total score while MPI version is second and Threads version is last

5 Conclusion

From the result, we can see a clear trend where the implementation of De Bruijn Graph in Threads and MPI are generally slower than the serial version of the algorithm.
When it comes to De Bruijn Graph Building Time for all test cases, In Test Case 1 and Test Case 2, we see that for the Threads version, the distribution of work between each threads are uneven but currently each threads are given equal partition of a vector of sequences (each sequence line of the file). But the time of each threads in Test Case 3, is pretty distributed. As for the MPI Version, since each machine works on constructing its own graph, the time should be on par with serial version, which we can see that is the case for all Test Cases.
As For the Eulerian Path Traversing Time, the Theads and MPI implementation all show longer time than serial version in all Test Cases. We can see that for Threads implementation, the time various between threads, this is because each thread will start at random thread that could result in an unbalanced load between threads.
Another reason for slower time for threading and MPI is most likely due to the fact that this algorithm had to deal with large amount of data in string causing a lot of overhead when ensuring shared data are safely handled in multithreading and when communications are made in MPI. Especially need to note on the significant increase in time in MPI version, other than the communication overhead, this algorithm also requires each machine in MPI to be "synchronized" so it doesn't create deadlock or result

in the wrong answer. Also, MPI is not made to communicate custom data structure or string type. So in order for necessary communication to be made, the program first need to convert string to char and back again and this will cause even more overhead. Another limitation of MPI is when handling MPI communication command like Bcast or Send, where if one machine exit the program first, other processes will wait indefinitely at Bcast, Send or other command causing a deadlock. The ensure the program still work, starting thread of each processes is not select at random, all ranks starts at first node and hope they will enter different path down the line, but all my test cases result in a graph that is pretty linear so all the time for MPI eulerian path time is similar.
Lastly, when looking at the number of nodes and edges for the first two cases, because the first two cases are not Eulerian path or cycle, their constructed graph consists of many subgraphs and depends on the starting point when traversing, it can result in various length. It is also worth noting that it is possible for serial and threading variation to result in segmentation fault with runs. The most likely reason for this is due to both of them using a recursive function and if the depth of recursion is too high, it could cause a stack overflow, which would result in a segmentation fault.

When it comes to the BLAST results of all three test case, although different implementation returns a different max score and total score, it is not as informative in this case as the score will change with different kmers. It is more important to compare the Query Cover, E value, and Percentage Identity and all three implementations are consisted in all those fields in all test cases.

To conclude, when it comes to De Bruijn Graph, performance is harder to be improved through the use of Threads and MPI due to the large amount of data that need to synchronized/communicated especially with MPI where its main focus is not on string, but the project does shows that it is possible to parallelize Hierholzer's algorithm through the parallelism approach where thread start at a head node and when a thread hits the head node of another thread, they join paths, and improved performance may be achieved with more better approach.

References

[1] Satam H, M. U. W. S. Z. G. R. S. T. R. B. S. M. A. D. G. M. S., Joshi K. Next-generation sequencing technology: Current trends and advancements. *Biology (Basel)* **12(7)**, 997 (2023).

[2] Cameron DL, P. J. D. H. M. R. D. A. S. T. P. A., Schröder J. Gridss: sensitive and specific genomic rearrangement detection using positional de bruijn graph assembly. *Genome Res* **27(12)**, 2050–2060 (2017).

[3] Li D, L. R. S. K. L. T., Liu CM. Megahit: an ultra-fast single-node solution for large and complex metagenomics assembly via succinct de bruijn graph. *Bioinformatics* **31(10)**, 1674–1676 (2015).

[4] Compeau PE, T. G., Pevzner PA. How to apply de bruijn graphs to genome assembly. *Nat Biotechnol* **29(11)**, 987–991 (2011).

[5] Medvedev P, P. M. What do eulerian and hamiltonian cycles have to do with genome assembly? *PLoS Comput Biol* **17(5)**, e1008928 (2021).

[6] Jamshed Khan, S. D. R. P., Marek Kokot. Scalable, ultra-fast, and low-memory construction of compacted de bruijn graphs with cuttlefish 2. *Genome Biol* **23**, 190 (2022).

[7] Břinda, K., Baym, M. & Kucherov, G. Simplitigs as an efficient and scalable representation of de Bruijn graphs. *Genome Biology* **22** (2021).

[8] Rayan Chikhi, G. R. Space-efficient and exact de bruijn graph representation based on a bloom filter. *Algorithms Mol Biol* **8**, 22 (2013).

[9] Simpson, K. W. S. D. J. J. E. S. S. J. J., Jared T. & Birol, I. Space-efficient and exact de bruijn graph representation based on a bloom filter. *Genome research* **19(6)**, 1117–1123 (2009).