Scalable Genomic Assembly through Parallel *de Bruijn* Graph Construction for Multiple K-mers

Kanak Mahadik, Christopher Wright, Milind Kulkarni, Saurabh Bagchi, Somali Chaterji
Purdue University, West Lafayette, IN, USA
{kmahadik,christopherwright,milind,sbagchi,schaterji}@purdue.edu

ABSTRACT

Extraordinary progress in genome sequencing technologies has led to a tremendous increase in the number of sequenced genomes. However, biologists have run into a computational bottleneck to assemble large and complex genomes quickly, due to the lack of scalable and parallel *de novo* assembly algorithms. Among several approaches to assembly, the iterative *de Bruijn* graph (DBG) assemblers, such as IDBA-UD, generate high-quality assemblies by sequentially iterating from small to large k-values used in graph construction. However, this approach is time intensive because the creation of the graphs for increasing k-values proceeds sequentially. For example, with just eight k-values, graph construction takes 96% of the total time to assemble a metagenomic dataset with 33 million paired-end reads. In this paper, we propose ScalaDBG, which transforms the sequential process of DBG construction for a range of k values, to one where each graph is built independently and in parallel. We develop a novel mechanism whereby the graph for the higher k value can be "patched" with contigs generated from the graph with the lower k value. We show that for a variety of datasets our technique can assemble complex genomes much faster than IDBA-UD (6.7X faster for the most complex genome in our dataset) while maintaining the same accuracy for the assembled genome. Moreover, ScalaDBG's multi-level parallelism allows it to *simultaneously* leverage the power of mighty server machines by using all their cores *and* of compute clusters by scaling out.

1 INTRODUCTION

With the rapid advancement of sequencing technologies projected to generate 2^{40} exabytes of data by 2025 just for the human genomes [16], there is a dire need for *de novo* assembly algorithms to play catch-up. A high latency genome assembly kernel, a fundamental step in all genomic analyses pipelines, is a bottleneck for the subsequent analyses kernels, negatively affecting the overall performance and impeding the extraction of knowledge from raw genomic datasets.

Currently the most popular de novo genome assembly method is the *de Bruijn Graph* (DBG) method [3], used by assemblers such as Velvet [17], ABySS [15], and ALLPATHS-LG [5].

ACM-BCB '17, August 20-23, 2017, Boston, MA,USA.

978-1-4503-4722-8/17/08…$15.00
DOI: http://dx.doi.org/10.1145/3107411.3107482

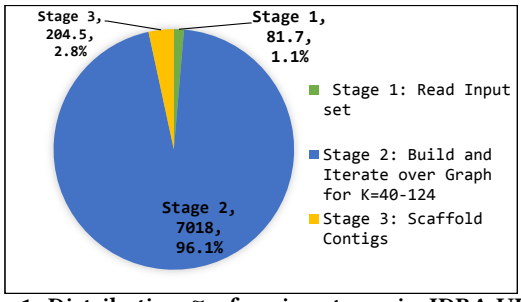

Figure 1: Distribution % of major stages in IDBA-UD, the time taken for each stage is provided in seconds for CAMI metagenomic dataset with 33 million paired-end reads of length 150, insert size 5kbp, 8 k-values : $40-124$, step-size 12.

Characteristics of the DBG structure are highly dependent on the value of k selected for graph construction, with smaller k-values giving rise to *branching* of graphs and larger k values resulting in fragmented graphs [1, 7, 10, 12]. To achieve superior assemblies than a DBG built with a single k-value IDBA (Iterative DBG Assembler) [10], IDBA-UD [12], SOAPdenovo2 [7], and SPAdes [1] use multiple k-values, iteratively, to assemble the sequenced reads. Essentially, a DBG with a small k-value can be traversed to infer what longer sequences *might look like*, allowing a DBG with a larger k-value to incorporate this additional information to "patch up" gaps.

Unfortunately, while these solutions can leverage multiple k-values to generate higher quality assemblies, the time taken for assembly increases linearly with the number of different k-values being used. Figure 1 shows the time taken by IDBA-UD to execute the key stages of assembly: Reading the sequence file (stage 1), processing with multiple k-values—first building the graph and then iterating over the graph with $k = 40 - 124$ with a step size of 12 to generate the contigs (stage 2), and then, scaffolding (stage 3) to get the final assembly. We invoked IDBA-UD on a metagenomic dataset part of the CAMI benchmark with 33 million paired-end reads of length 150 and insert-size 5kbp. We used an Intel Xeon E5-2670 2.6 GHz node with 16 cores and 32 GB memory for the experiment. As shown in Figure 1, we found that the stage 2 of iterative graph-construction process, comprising of initial graph-build construction and iteration takes up 96.1% of the total workflow time. Clearly, the iterative graph-construction step dominates the total time taken for assembly.

The most common metric to assess assembly quality is *N50*[1]. We see in Table 1 that as the number of k-values in the range set is increased, the quality of the assembly improves. However,

[1]N50 is defined as the length of the smallest contig above which 50% of an assembly would be represented (or smallest scaffold if it is applied after scaffold construction). A higher N50 number indicates a better assembly.

k-value range (count)	step size	Total Time (sec)	N50	(%) improvement in N50	(%) increase in execution time
40-124 (2)	84	3165	3547	-	-
40-124 (4)	28	4954	8255	132	56.5
40-124 (8)	12	7100	10729	202	124.3

Table 1: Relationship between k-values, Quality of Assembly, and Runtime for IDBA-UD running CAMI medium-complexity metagenomic dataset

the total time taken also increases proportionately, *i.e.*, linearly in proportion with the number of different k values being used. This is an undesirable consequence of iterating through increasing numbers of k-values *sequentially*. Thus, although the N50 value of the assembly, when iterating using k-values of 40-124, with a step-size of 12, is 3X that of using k-values 40 and 124, it takes 124.3% higher time for constructing the final graph with these k-values.

To address this concern, we propose *ScalaDBG*, a new parallel assembly algorithm, which parallelizes stage 2 of Figure 1. The key insight behind ScalaDBG is that the graphs for multiple k-values need not be constructed serially. Instead, each graph construction can be done independently and in parallel. Accumulating the graph for the higher k-value, such as in IDBA-UD, introduces an apparent dependency on the graph with lower k-values. We remove this dependency, by introducing a *patching technique*, which can patch the higher k-valued graph (k_2) with contigs from the lower k-valued graph (k_1). Crucially, the first stage of construction of the k_1 graph and the k_2 graph can proceed in parallel and the relatively shorter stage of patching the k_2 graph with the contigs from the k_1 graph happens subsequently. Thus, the gaps in the higher k-valued graph are *cemented* from the contigs of the lower k-valued graph, with the branches of the lower k-valued graph removed. For a long list of k-values, ScalaDBG employs the divide-and-conquer approach according to the *tree reduction* pattern and identifies greater scope for parallel execution . ScalaDBG's technique is general and can be applied to parallelize any DBG-assembler that performs either single (Velvet) or multi-k value (IDBA, SPAdes) DBG construction, due to its modular design.

Several steps within the construction of a single DBG, such as k-mer counting, indexing, and lookup, occur in parallel. We borrow IDBA-UD's strategy for parallelizing graph construction for a *single* k-value. Thus, ScalaDBG exploits parallelism at 2 levels—a coarse-grained level (by constructing graphs with multiple k-values in parallel) and a fine-grained level (parallelizing the construction of each individual DBG). Thus, ScalaDBG can leverage the power of a robust server, with multiple processors and a large RAM, for *vertical scaling or scale-up*, as well as a cluster with multiple nodes, for *horizontal scaling or scale-out*. As a sample result, consider that IDBA-UD assembles the CAMI metagenomic dataset for a k-value range of $40-124$, with a step of 12 in ≈2.2 hours on a single node. With ScalaDBG, we assemble the same dataset in ≈1.1 hours on 8 such nodes, which is 2X faster.

We make the following technical contributions in this paper:

(1) We break the dependency in DBG creation for multiple k-values—from a purely serial process to one where the most time-consuming part (the DBG creation for individual k-values) is parallelized. This innovation can be applied out-of-the-box to most DBG-based assemblers.
(2) We develop a divide-and-conquer strategy for handling a long chain of k-values while efficiently utilizing all available machines in a cluster and all available cores on a machine.
(3) We develop a software package ScalaDBG that uses OpenMP for scale-up within one server and MPI for scale-out across multiple servers. The software package is available through https://github.com/purdue-dcsl/Scaladbg.

2 BACKGROUND

In this section, we provide background on the creation of DBGs using multiple k values and an overview of IDBA-UD.

2.1 Using Multiple k-values in Iterative Graph Assembly

Figure 2 shows the effect of using a small k ($k = 3$), and a larger k ($k = 4$) during DBG construction. Figure 2 (a) shows the graph constructed from read set with $k = 3$. The vertices are consecutive 3-mers of the read set. They are connected to each other if they have a 2-mer overlap. This graph has branching at vertex ACG due to repeating region in the genome *ACGT* and *ACGA*. The contig set generated by identifying maximal paths in the graph is *{AATGCCGT,ACGAA,ACGT,CGTACG}*. As the value of k is increased to 4, the branch disappears as the higher k-value can now distinguish between the repeat region in *ACGT* and *ACGA*. However, some reads such as *CCGTA* and *GTACG* are are not sampled from the genome sequence and so vertices and edges in the graph are missed. Hence *GTAC* and *TACG* cannot be connected. Thus, Figure 2 (b) with $k = 4$ has gaps in it. In general, DBG built using a lower k value has multiple branches, and DBG built using a higher k value has gaps. The contig set generated for $k = 4$ is *{TACGTAC, TACGAA, AATGCCGT}*. If we can take the graph built with $k = 4$ and augment it with the contigs from the $k = 3$ graph, then it is conceptually possible to arrive at the graph shown in Figure 2(c). In Figure 2(c), an edge is added between circled vertices $GTAC$ and $TACG$ due to presence of the substring *GTACG* in the contig set obtained using $k = 3$. After adding the edge, and traversing the composite graph, contigs longer than those created from both $k = 3$ and $k = 4$ are obtained : *{TACGTAC***G**,*TACGAA,AATGCCGT*}.

2.2 IDBA-UD

IDBA-UD is a de Bruijn graph assembler. It iterates on a range of k values from $k = k_{min}$ to $k = k_{max}$, with a step-wise increment of s. It maintains an accumulated de Bruijn graph H_k at each step. In the first step a de Bruijn graph $G_{k_{min}}$ is generated from the input reads. For $k = k_{min}$, H_k is equivalent to $G_{k_{min}}$. At any step, contigs for graph H_k are generated by considering all maximal paths in graph H_k. All vertices in any maximal path have an in-degree and out-degree equal to 1 except the vertices at the start and end of the path. All reads from the input set that are substrings of these contigs are removed, reducing the input size at each step. A read of length r generates $r - k + 1$ vertices. As k is increased each read will introduce fewer vertices. This reduction in input read set coupled with the fact that there are fewer vertices for larger k values, makes subsequent graph constructions less time consuming.

The inputs to the next step, where $k = k_{min} + s$ consist of the graph H_k, the remaining reads, and the contigs from H_k. Each path

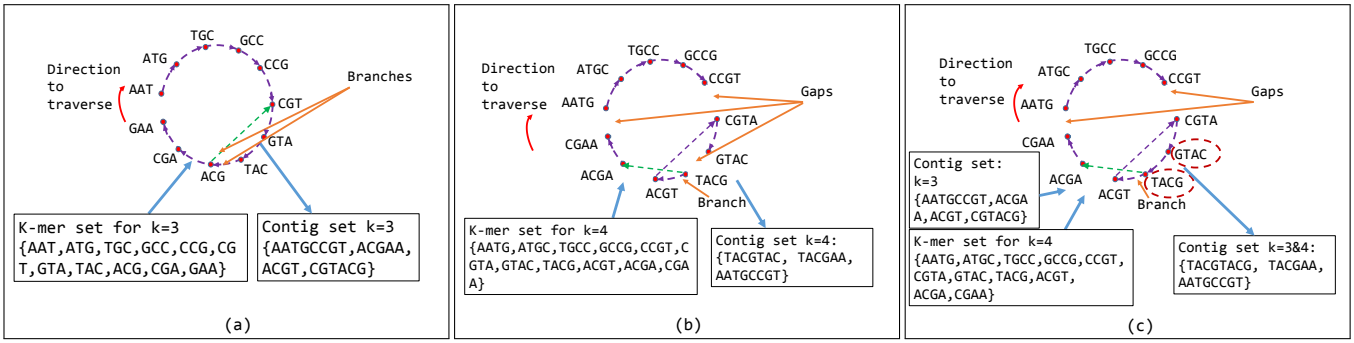

Figure 2: Desired Genome Sequence : AATGCCGTACGTACGAA, Read Set : *AATGC, ATGCC, GCCGT, TGCCG, CGTAC, TACGT, ACGTA, TACGA, ACGAA* De Bruijn Graph for $k = 3$ (sub-figure (a)) and $k = 4$ (sub-figure (b)). The final graph (sub-figure (c)) can be created by filling in some of the gaps in the $k = 4$ graph with contigs from the $k = 3$ graph. The vertices for which new edge is added (sub-figure (a)) are circled. Traversing this final graph results in the final contig set.

of length s in H_k is converted to a vertex. All such vertices are connected by an edge if the corresponding $(k + s + 1)$-mer exists in either the remaining reads or the contigs of H_k. This process is repeated for each subsequent iteration until $k = k_{max}$ is reached. Note that in this algorithm at iteration i, graph $H_{k_{min}+i*s}$ depends on graph $H_{k_{min}+(i-1)*s}$ obtained at iteration $i-1$, the reduced read set, and the contigs obtained at the iteration $i-1$.

This dependency forces IDBA-UD to operate sequentially on the chain of k-values, no matter how long the chain is. This is the crux of the problem that we address through ScalaDBG.

IDBA-UD also uses existing graph simplification strategies such as dead end removal and merging bubbles to get longer final contigs. In its dead end removal phase, IDBA-UD removes short simple paths leading to dead ends. In the bubble merging phase, several similar sequences are merged into one sequence. A *bubble* is formed when several similar paths have the same start and end vertex [17]. These bubbles can be formed due to errors in reads, generating similar paths with few differences. Bubble merging and dead end removal phases throw away vertices and edges from the graphs for simplification. At the end, scaffolding techniques are applied to the contigs of H_{kmax} to get the final set of contigs.

3 DESIGN OF SCALADBG

In this section we describe the design details of ScalaDBG, considering the build phase (building a DBG) and the patch phase (patching a partial DBG with contigs from a lower k-value DBG).

3.1 Build Phase

To simplify the exposition, we describe our protocol first using just two different k values, $k1$ and $k2$, with $k1 < k2$. Figure 3 shows the stages of ScalaDBG for the two k values. In the build phase, DBGs are built for each k value in parallel, and the build module generates G_{k1} and G_{k2}. The graph construction is the most time consuming phase of the entire pipeline, with the construction of G_{k1} taking the most time.

Vertices in G_{k1} and G_{k2} are all $k1$-mers and $k2$-mers obtained from the original input read set I, respectively. Vertices and edges are established in G_{k1} and G_{k2} as per the definition of DBG. We denote the number of vertices in G_{k1} and G_{k2} as $|G_{k1}|$ and $|G_{k2}|$

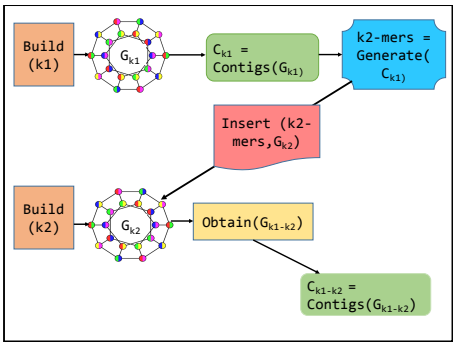

Figure 3: High Level Architecture Diagram of ScalaDBG. This shows the graph construction with only two different k values, $k1$ and $k2$ with $k1 < k2$. The graph G_{k2} is "patched" with contigs from G_{k1} to generate the combined graph G_{k1-k2}, which gives the final set of contigs. Different modules in ScalaDBG are highlighted by different colors.

respectively. Graph G_{k1} will typically be larger in size, in terms of vertices and edges, *i.e.*, $|G_{k1}| > |G_{k2}|$, since $k1 < k2$. Importantly, the creation of the DBGs for the two different k-values proceeds in parallel, unlike in all prior protocols. Now G_{k1} will have a higher number of branches than G_{k2}, while G_{k2} will have a higher number of gaps than G_{k1}. We generate contigs C_{k1} from G_{k1} by finding maximal paths according to standard practice (we use the IDBA algorithm specifically) but we do *not* yet proceed to generate C_{k2} from G_{k2}.

3.2 Patch Phase

We cannot simply create contigs from graph G_{k2} because it will have gaps relative to the graph G_{k1}. Therefore, our idea is to *patch* graph G_{k2}, *i.e.*, fill the gaps in the graph by bringing in more vertices and connecting the new vertices plus the existing vertices with additional edges. *The fundamental insight that we have is that contigs C_{k1} have the information to close some of these gaps.* We process C_{k1} by generating $k2$-mers from sequences in C_{k1}, *i.e.*, we generate all subsequences of length $k2$ from all contigs in the C_{k1} set. These $k2$-mers are inserted into the G_{k2} graph as vertices. The introduction of new vertices in the graph can result in the introduction of new

edges. Two vertices u and v in the graph G_{k2} are connected by an edge if the last $(k2-1)$ nucleotides of the $k2$-mer represented by u are the same as the first $(k2-1)$ nucleotides of the $k2$-mer represented by v, and u and v are consecutive k-mers in the contig set C_{k1} or read set. The resulting graph is denoted as G_{k1-k2}. Thus, G_{k1-k2} is the aggregated graph obtained by filling gaps of G_{k2} using C_{k1}. The final contig set is generated from G_{k1-k2}.

3.3 Patching Multiple k Values in Parallel

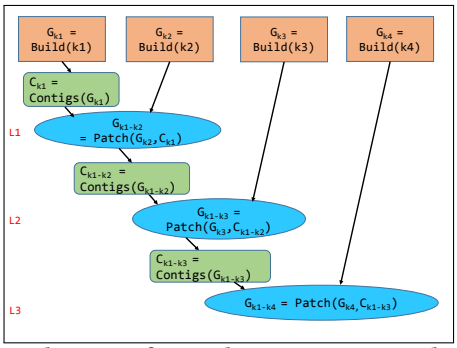

Figure 4: Schematic for ScalaDBG using serial patching, called ScalaDBG-SP

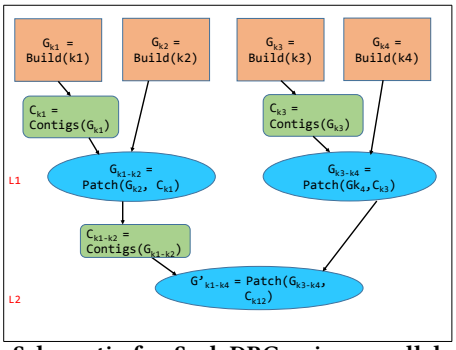

Figure 5: Schematic for ScalaDBG using parallel patching, called ScalaDBG-PP.

When the number of k values to be iterated over is greater than 3, *ScalaDBG has two options while patching*. It can adopt either *a serial method* shown in Figure 4, or *a parallel method* shown in Figure 5. Figure 4 shows the serial patching process when there are four different k values, $k1$, $k2$, $k3$, $k4$, and $k1 < k2 < k3 < k4$. Initially, graphs for each of the 4 k values are generated in parallel. In the serial variant of ScalaDBG, the graph associated with the lowest k value, $k = k1$ is assembled, and contigs are generated from the graph G_{k1}. The obtained contigs are used to patch the graph associated with the next higher k value ($k2$). Contigs are generated using the higher k valued graph. This process is repeated, serially for each increasing higher k value, until the graph associated with the highest k value in the chain is patched and assembled. The final set of contigs is generated from this final patched graph. Thus, the graph building for each separate k value occurs in parallel but the patching and generating contigs occurs serially. The advantage of the serial patching method is its simplicity, owing to the simple communication patterns between processes operating over the different k values. However, as the number of different k values increases, the number of serialized patching steps grows *linearly* with it. The serialized patching process starts dominating the total time of the workflow for ScalaDBG. To resolve this bottleneck, in the patch phase, ScalaDBG in the parallel mode allows multiple patch operations to occur in parallel.

The insight behind our parallel method is that multiple patch operations, which are independent of each other, can proceed in parallel. Thus, the patching process proceeds like a reduction tree. Figure 5 shows the parallel method of patching, where multiple patch processes occur in parallel, *e.g.*, the patching of graph G_{k2} with contigs from graph G_{k1} proceeds in parallel with the patching of graph G_{k4} with contigs from graph G_{k3}. Each pair of adjacent graphs is patched to generate a single graph. This process is repeated until there is only a single graph. Thus, patching proceeds according to the *tree reduction* parallel pattern. In this way, a long chain of k-values can be broken down, with both graph construction and patching happening in parallel. Thus, as the list of k values grows longer, ScalaDBG identifies greater scope for parallel construction while making use of more compute nodes. In the parallel patch method, the number of serialized patching steps grows only *logarithmically* with the number of different k values.

In the rest of the sections, we refer to ScalaDBG using serial patching as *ScalaDBG-SerialPatch* (or ScalaDBG-SP), and ScalaDBG using parallel patching as *ScalaDBG-ParallelPatch* (or ScalaDBG-PP). Between the serial and parallel patch methods, different pairs of graphs are merged with each other. Hence, the final contigs generated by the two methods may differ. In general, the assembly quality of the serial method is higher since the difference between k values associated with adjacent graphs is smaller and it has been shown that small jumps in the k-values leads to better quality aggregated DBGs [10].

4 CORRECTNESS OF SCALADBG METHODOLOGY

Theorem 4.1 (Equivalence between final graph obtained in iterative IDBA-UD and ScalaDBG- SP). *For a fixed iteration set of k-values starting from $k = kmin$ to $k = kmax$, the final graph obtained by ScalaDBG-SP and IDBA-UD is identical.*

Proof. We use the principle of Mathematical Induction to establish the equivalence of the resulting graph in ScalaDBG- SP and IDBA-UD.

Initial Step Let R represent the initial read set input to ScalaDBG-SP and IDBA-UD. When $kmax = kmin$, the final graph obtained after the first iteration is the final graph. There is no patching involved. ScalaDBG-SP and IDBA-UD generate $kmin-1$-mers from R and use the same build procedure to generate graph G_{kmin} which is the final graph.

Inductive Step For $kmax = K$, where $K > kmin$, the graph obtained using ScalaDBG-SP is identical to IDBA-UD. We denote this graph by G_K. We must prove, the statement is true for $kmax = K+1$.

The graph obtained by ScalaDBG-SP after constructing graphs from $k = kmin$ to $k = K$ in parallel, followed by serial patching is G_K. ScalaDBG-SP patches the graph G_{K+1} using contigs generated from G_K to get final graph P_{K+1}. IDBA-UD generates G_K at the end of $k = K$ iteration. After the next $K+1$ iteration, it generates H_{K+1} as the final graph. We need to show that $P_{K+1} = H_{K+1}$.

As explained in Section 2, in IDBA-UD, to construct H_{K+1} from G_K , first potential contigs in G_K are constructed by identifying *maximal paths*. Let the contig set of G_K be denoted by C_G_K and R_K represent the read set of IDBA-UD at beginning of iteration $K+1$. All reads in R_K that are substrings of a contigs in set C_G_K are removed. Let this new read set be denoted by R_{K+1}. Thus, $\{R_{K+1} = R_K - r\}, \forall r \in R_K$, r is a substring of a contig in C_G_K. In the construction of H_{K+1}, only the reads in R_{K+1} and the potential contigs of G_K stored in C_G_K are considered. H_{K+1} consists of vertices formed using edges in G_K where each edge (vi, vj) in G_K is converted into a vertex, representing a $(K+1)$-mer, if the $(K+1)$-mer is a substring of a contig in C_G_K.

ScalaDBG-SP starts with the original read set R and generates all $(K+1)$-mers of the reads. Graph G_{K+1} is built using R. Contig set C_G_K (same graph will generate same contig set, inductive step) is built using G_K. Then ScalaDBG-SP extracts $K+1$-mers from each contig in the set C_G_K and inserts it into G_{K+1}. Vertices and edges in H_{K+1} in IDBA-UD obtained by upgrading vertices in G_K are $(K+1)$-mers of contigs in C_G_K. Hence ScalaDBG-SP is guaranteed to insert these vertices and edges as we create $(K+1)$-mers from the contigs. All vertices and edges in P_{K+1} are formed using the original read set R and the contigs of G_K. Now R_{K+1} is a proper subset of R. So all vertices and edges inserted using R_{K+1} will be inserted by R. Hence all vertices and edges formed using R_{K+1} + C_G_K in H_{K+1} will be formed using R + C_G_K in P_{K+1}. Thus P_{K+1}, formed by patching together G_K and G_{K+1} is identical to H_{K+1}. From *Initial Step*, *Inductive Step* and principle of mathematical induction, IDBA-UD and ScalaDBG- SP generate identical graphs. □

Although the process of build and serial patch process of ScalaDBG essentially generates a graph identical to IDBA-UD, the assembly metrics of ScalaDBG- PP, ScalaDBG-SP and IDBA-UD differ. There are primarily two reasons for the differences : i) the out of order patching in ScalaDBG-PP and ii) the graph simplification procedures (bubble merging and dead end removal).

The order in which graphs built with different k values is significantly different for ScalaDBG- PP than ScalaDBG-SP, and IDBA-UD. While IDBA-UD iterates over the k values in an increasing order, with a maximum of *step-size* difference between the iterations, ScalaDBG-PP follows tree-style reduction to patch the graphs. Hence the difference in k values of two patched graphs will be higher, and the order of patching will also be different. The difference in output of ScalaDBG-SP, ScalaDBG-PP and IDBA-UD is also due to the graph simplification procedures applied before generating the contigs. The bubble merging and dead-end removal phases remove incorrect vertices and edges based on their multiplicity. The graphs obtained by ScalaDBG and IDBA-UD workflows have different multiplicity information for their vertices and edges. However, in our evaluation section, we will show that the difference is not statistically significant.

5 IMPLEMENTATION

We implement ScalaDBG using OpenMP (Version 4.0) (for parallelism within a node) and MPI (MVAPICH 2.2) (for parallelism across nodes in a cluster), compiled with GCC version 4.9.3.

In ScalaDBG- PP the work is split up into n chunks, where $n = number\_of\_kmers$. Each MPI process computes the $kmer\_size$ it will work on, and read the input files to build the DBG. The worker processes compute and send their DBGs to the Master. The Master process receives all other graphs and patches them in serial order, as shown at a high level in Figure 4. Intermediate graph representations are written and read from the Lustre Parallel File System.

In ScalaDBG- PP, the process is the same as ScalaDBG-SP, up through building a DBG. For the next stage we split up the processes with half receiving and half sending the DBG as shown in Figure 5. After a process sends its DBG, it is no longer used in the patching. This stage is repeated until there is only 1 process that receives, which will always be the Master. The Master will then complete the last assembly and perform contig generation.

6 EVALUATION

In this section we evaluate ScalaDBG in comparison to IDBA-UD in terms of the time taken to perform the assembly and the quality of the assembly. We used two different types of sequencing read sets - metagenomic and single cell sequencing, as they are typically assembled by the genomics community using the iterative de-bruijn graph approach.

6.1 Evaluation Setup and Data Sets

We performed our experiments on an Intel Xeon Infiniband cluster. Each node had Intel Xeon E5-2670, 2.6 GHz with 16 cores per node, and 32 GB of memory. The nodes were connected with QDR Infiniband. We used the latest version of IDBA-UD (1.1.1)[12]. We used the read sets listed in Table 2. We obtained the *S. aureus* and SAR 324 single-cell datasets from [9]. The metagenomics dataset was obtained from the CAMI benchmark [13]. In the case of metagenomic and single cell sequencing datasets, sequencing depths of different regions of a genome, or genomes from different organisms are exceedingly uneven. Hence multiple k-values are required for accurately assembling the datasets. So we evaluate ScalaDBG and IDBA-UD using these relevant datasets. The number of nodes in the cluster are equal to the number of different k-values in the configuration for ScalaDBG, while IDBA-UD can only run on a single node. ScalaDBG outputs contigs for a given dataset. Existing scaffolding techniques can be applied to these output contigs to get the final assembly. We only focus on evaluating the performance and accuracy metrics of the assembled contigs in the following experiments for IDBA-UD and ScalaDBG since our contribution is confined until that stage.

Name	Read Set Type	Read Length	# of reads	Characteristics
RM2	Real, Metagenomic	150 bp	33140480	PE,Insert size:5 kbp
SC -*S. aureus*	Real, Single Cell	100 bp	66,997,488	PE,Insert size:214bp
SC-SAR324	Real, Single Cell	100 bp	55,733,218	PE,Insert size:180bp

Table 2: Read Sets used in the Experiments. PE denotes Paired End reads

6.2 Performance Tests

Figures 6, 7, 8, and 9 show the time taken by IDBA, ScalaDBG-SP, and ScalaDBG-PP to generate contigs from the 3 different read sets mentioned in Table 2. The tests bring out the effect of different k-values on performance. For the metagenomic dataset we changed the step size to get three different configurations. We used step sizes of 28, 12, and 6 in the range $40 - 124$ (we give step sizes in

reverse order because this corresponds to an increasing number of k-values). For the single cell datasets, we used step sizes of 14, 6, and 3 in the range of $29 - 71$. The lower and upper bounds of the range is lower for the single cell dataset since its reads are shorter in length. We also ran SAR324 in the range of $20 - 50$ with step of 10, 5, and 2 to get 4, 7, 16 k values respectively.

The first 3 configurations have successively higher number of k-values: 4, 8,and 15, and are meant to evaluate the effect on quality of assembly and running time as the number of k values is increased. In addition the range extremes are held constant to obtain the information obtained from the two extreme k values. We ran ScalaDBG by matching the number of nodes in the cluster with the number of distinct k-values in the configuration to maximize scaling out performance for ScalaDBG. IDBA-UD on the other hand can only run on a single node. We report the overall execution times for both IDBA and ScalaDBG for generating the final contigs from the input readset.

We see that the speedup for ScalaDBG-PP and ScalaDBG-SP over IDBA increases with increase in the number of k values, for all the read sets. Speedup of ScalaDBG completely depends on the specific k values chosen. For the SAR324 dataset in the range of 20-50 with step size of 2, speedup of ScalaDBG- PP is 6.7X and speedup of ScalaDBG-SP is 3.1X over IDBA-UD. Of all the remaining readsets and configurations, ScalaDBG-PP achieves a maximum speedup of 3.3X for the SC-SAR324 readset in the $\{29 - 71\}$, step size 3 configuration. ScalaDBG- SP achieves a maximum speedup of 1.6X for RM2, SC-*S.aureus*, and SC-SAR324 readsets in the configurations processing 15 k values. For all the datasets and configurations, ScalaDBG is faster than IDBA-UD. Further, ScalaDBG-PP is faster than ScalaDBG-SP since it has higher parallelism during assembly for the patching process. Speedup of ScalaDBG-SP and ScalaDBG-PP over IDBA is higher for the larger readsets of RM2, SAR 324, and *S. aureus*.

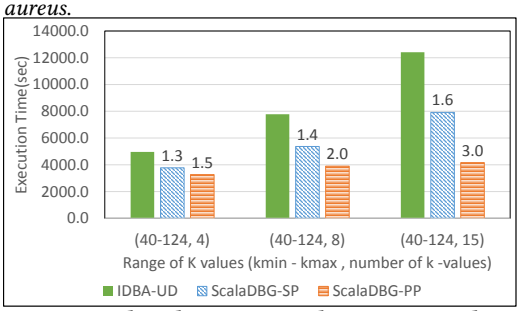

Figure 6: Time taken by IDBA, ScalaDBG-SP, ScalaDBG-PP on RM2 data set.

6.3 Accuracy

Table 3 show the accuracy metrics for assembling the datasets in Table 2 for the above performance tests. For the metagenomic dataset and SAR324 we only reported number of contigs, N50 and max contig length. The metagenomic dataset reference assemblies contain multiple genomes, while for SAR324 we could not obtain its reference genome, so we could not report the coverage, NGA50, and number of missassemblies for them. For SC-*S.aureus*, both IDBA-UD and ScalaDBG have NGA50 of 26379 and 3 missassemblies, coverage is 98.1 for IDBA-UD and 98.2 for ScalaDBG in all three configurations. Differences in the accuracy arise as a result of the

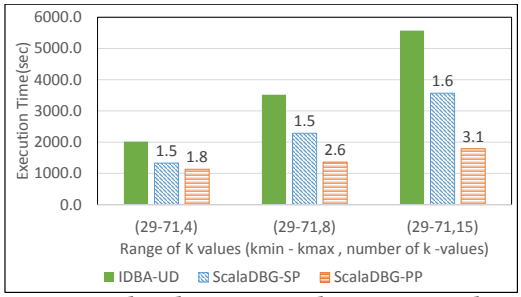

Figure 7: Time taken by IDBA, ScalaDBG-PP, ScalaDBG-PP for completing assembly on the SC-*S.aureus* dataset.

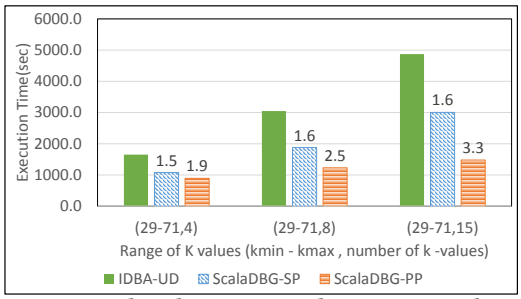

Figure 8: Time taken by IDBA, ScalaDBG-PP, ScalaDBG-PP for completing assembly on the SC-SAR324 dataset.

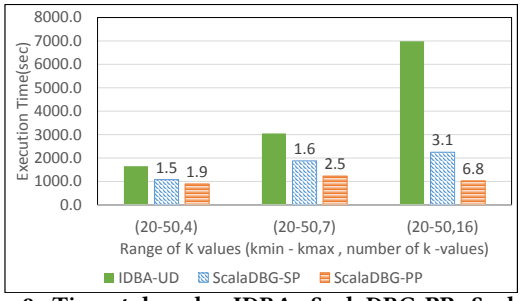

Figure 9: Time taken by IDBA, ScalaDBG-PP, ScalaDBG-PP for completing assembly on the SC-SAR324 dataset for range(20-50)

invocations of graph simplification and out of order patching, explained in Section 4. The results demonstrate that ScalaDBG and IDBA have comparable accuracy metrics in all cases. We performed the T-test and determined that the differences in the assembly metric N50 obtained for ScalaDBG-SP, ScalaDBG-PP and IDBA-UD are not statistically significant.

6.4 Scalability Tests

To evaluate the scaling out for ScalaDBG, we used the metagenomic dataset RM2. We varied the k values for ScalaDBG in the range of $\{40 - 124\}$ with a step size of 6. The range has 15 k-values. We increase the number of nodes in the cluster from 1 to 15, and measure the speedup achieved as shown in Figure 10. As can be seen, ScalaDBG achieves a speedup of 3X for the RM2 dataset , compared to the baseline version running on 1 node. It scales at nearly constant efficiency(slope of speedup curve) upto 8 nodes. The reduction in efficiency at 15 nodes is due the slightly imbalanced

Assembler	# Contigs	N50 (bp)	Max Contig Length	# Contigs	N50 (bp)	Max Contig Length
	RM2 k=40-124,4			SC-*S.aureus* k=40-124,4		
IDBA-UD	123807	2251	572031	400	24855	126604
ScalaDBG-SP	122427	2281	571953	377	24855	126604
ScalaDBG-PP	123037	2249	444176	370	24855	126604
	RM2 k=40-124,8			SC-*S.aureus* k=40-124,8		
IDBA-UD	121911	2457	563546	412	24855	126604
ScalaDBG-SP	121955	2582	573903	384	24855	126604
ScalaDBG-PP	121772	2408	444517	373	24855	126604
	RM2 k=40-124,15			SC-*S.aureus* k=40-124,15		
IDBA-UD	121720	2504	563546	413	24855	126604
ScalaDBG-SP	119814	2679	573903	393	24855	126604
ScalaDBG-PP	121568	2472	444518	374	24855	126604
	SC-SAR324 k=29-71,4			SC-SAR324 k=20-50,4		
IDBA-UD	733	61419	202281	1082	32119	131087
ScalaDBG-SP	709	64747	202281	1085	38257	131546
ScalaDBG-PP	705	62374	202281	1080	38257	131546
	SC-SAR324 k=29-71,8			SC-SAR324 k=20-50,7		
IDBA-UD	742	60700	202281	1088	33192	131087
ScalaDBG-SP	710	64747	202281	1087	38257	131546
ScalaDBG-PP	703	63904	202281	1078	38257	131546
	SC-SAR324 k=29-71,15			SC-SAR324 k=20-50,16		
IDBA-UD	747	60700	202281	7740	22977	131087
ScalaDBG-SP	723	64747	202281	8342	24254	131041
ScalaDBG-PP	712	64795	161406	8118	24254	131041

Table 3: Accuracy Comparison for Performance Tests on RM2, SC-*S.aureus* and SC-SAR324 datasets

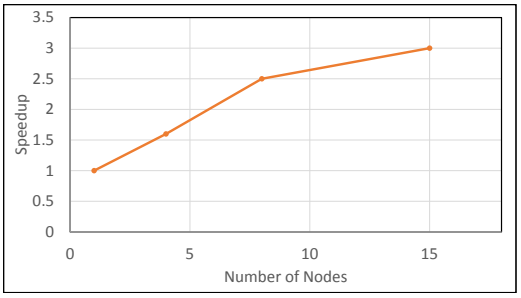

Figure 10: Scaling Results for ScalaDBG, speedup shown w.r.t ScalaDBG running on 1 node

parallel reduction tree of ScalaDBG-PP. The speedup demonstrates that ScalaDBG can scale out in a cluster and leverage the power of all its cores.

7 RELATED WORK

Efficient de-novo assembly applications have been proposed to deal with the tremendous increase in genomic sequences [1, 2, 4–8, 10–12, 14, 15, 17]. These assembly applications are either limited to scaling up on a single node, or cannot use multiple k values during the process of assembly. To our knowledge, there has been no previous work on parallelizing de bruijn graph construction for multiple K-mers on multiple nodes in a cluster.

Ray [2], ABySS [15], PASHA [6], and HipMer [4] can parallelize the assembly process for a single k value on multiple nodes in a cluster. However, they do not generate good quality assemblies for metagenomic and single cell datasets with uneven sequencing depths, unlike ScalaDBG. SGA [14], Velvet [17], SOAPdenovo [7], ALLPATHS-LG [5] can parallelize the assembly process on multiple cores of a single node. Metagenomic assemblers such as Meta-velvet [8] also do not use multiple k values in assembly. IDBA, IDBA-UD, and SPAdes can utilize multiple k values. They are limited to scale on a single node. ScalaDBG can scale up, scale out, and also process multiple k-values.

8 CONCLUSION

Faster and cheaper sequencing technologies have led to a massive increase in the amount of sequencing data. Efficient assembly algorithms are key to uncovering knowledge within the data and make possible medical breakthroughs based on single cell and metagenomic datasets. Existing iterative methods of debruijn graph construction such as IDBA-UD, generate longer contigs but are completely sequential and suffer from significantly longer graph construction times. In this paper we presented a technique ScalaDBG that breaks this serial process of graph construction into a parallel process. Our technique is general, and can be easily extended to other DBG-based assembly algorithms.

REFERENCES

[1] Anton Bankevich, Sergey Nurk, Dmitry Antipov, Alexey A Gurevich, Mikhail Dvorkin, Alexander S Kulikov, Valery M Lesin, Sergey I Nikolenko, Son Pham, Andrey D Prjibelski, and others. 2012. SPAdes: a new genome assembly algorithm and its applications to single-cell sequencing. *Journal of Computational Biology* 19, 5 (2012), 455–477.

[2] Sébastien Boisvert, François Laviolette, and Jacques Corbeil. 2010. Ray: simultaneous assembly of reads from a mix of high-throughput sequencing technologies. *Journal of computational biology* 17, 11 (2010), 1519–1533.

[3] Phillip EC Compeau, Pavel A Pevzner, and Glenn Tesler. 2011. How to apply de Bruijn graphs to genome assembly. *Nature biotechnology* 29, 11 (2011), 987–991.

[4] Evangelos Georganas, Aydın Buluç, Jarrod Chapman, Steven Hofmeyr, Chaitanya Aluru, Rob Egan, Leonid Oliker, Daniel Rokhsar, and Katherine Yelick. 2015. HipMer: an extreme-scale de novo genome assembler. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 14.

[5] Sante Gnerre, Iain MacCallum, Dariusz Przybylski, Filipe J Ribeiro, Joshua N Burton, Bruce J Walker, Ted Sharpe, Giles Hall, Terrance P Shea, Sean Sykes, and others. 2011. High-quality draft assemblies of mammalian genomes from massively parallel sequence data. *Proceedings of the National Academy of Sciences* 108, 4 (2011), 1513–1518.

[6] Yongchao Liu, Bertil Schmidt, and Douglas L Maskell. 2011. Parallelized short read assembly of large genomes using de Bruijn graphs. *BMC bioinformatics* 12, 1 (2011), 354.

[7] Ruibang Luo, Binghang Liu, Yinlong Xie, Zhenyu Li, Weihua Huang, Jianying Yuan, Guangzhu He, Yanxiang Chen, Qi Pan, Yunjie Liu, and others. 2012. SOAPdenovo2: an empirically improved memory-efficient short-read de novo assembler. *Gigascience* 1, 1 (2012), 18.

[8] Toshiaki Namiki, Tsuyoshi Hachiya, Hideaki Tanaka, and Yasubumi Sakakibara. 2012. MetaVelvet: an extension of Velvet assembler to de novo metagenome assembly from short sequence reads. *Nucleic acids research* 40, 20 (2012), e155–e155.

[9] University of California at San Diego. 2011. Single cell data sets. http://bix.ucsd.edu/projects/singlecell/nbt_data.html. (2011).

[10] Yu Peng, Henry CM Leung, Siu-Ming Yiu, and Francis YL Chin. 2010. IDBA–a practical iterative de Bruijn graph de novo assembler. In *Annual International Conference on Research in Computational Molecular Biology*. Springer, 426–440.

[11] Yu Peng, Henry CM Leung, Siu-Ming Yiu, and Francis YL Chin. 2011. Meta-IDBA: a de Novo assembler for metagenomic data. *Bioinformatics* 27, 13 (2011), i94–i101.

[12] Yu Peng, Henry CM Leung, Siu-Ming Yiu, and Francis YL Chin. 2012. IDBA-UD: a de novo assembler for single-cell and metagenomic sequencing data with highly uneven depth. *Bioinformatics* 28, 11 (2012), 1420–1428.

[13] Alexander Sczyrba, Peter Hofmann, Peter Belmann, David Koslicki, Stefan Janssen, Johannes Droege, Ivan Gregor, Stephan Majda, Jessika Fiedler, Eik Dahms, and others. 2017. Critical Assessment of Metagenome Interpretation- a benchmark of computational metagenomics software. *bioRxiv* (2017), 099127.

[14] Jared T Simpson and Richard Durbin. 2012. Efficient de novo assembly of large genomes using compressed data structures. *Genome research* 22, 3 (2012), 549–556.

[15] Jared T Simpson, Kim Wong, Shaun D Jackman, Jacqueline E Schein, Steven JM Jones, and Inanç Birol. 2009. ABySS: a parallel assembler for short read sequence data. *Genome research* 19, 6 (2009), 1117–1123.

[16] Zachary D Stephens, Skylar Y Lee, Faraz Faghri, Roy H Campbell, Chengxiang Zhai, Miles J Efron, Ravishankar Iyer, Michael C Schatz, Saurabh Sinha, and Gene E Robinson. 2015. Big data: astronomical or genomical? *PLoS Biol* 13, 7 (2015), e1002195.

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